走进搜索引擎

STEPPING INTO SEARCH ENGINE

潘雪峰　花贵春　梁　斌　　编著

（第2版）

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

走进搜索引擎

（第 2 版）

潘雪峰　花贵春　梁斌　编著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书由搜索引擎开发研究领域三位年轻的博士生精心编写，作者们希望将自己对搜索引擎的理解和实际应用相结合，让未接触过搜索引擎原理和方法的读者也能轻松读懂该书的大部分内容。

本书在第 1 版的基础上，删除了搜索引擎历史等章节，并对错误和不足进行了修订和补充，同时增加了潘雪峰编写的第 6 章“搜索引擎日志分析”，花贵春编写的第 7 章“排序学习（Learning to Rank）”和梁斌编写的第 8 章“搜索引擎的性能调优”三个主要章节，变更的内容约占第 1 版的一半。

本书作为搜索引擎原理与技术的入门书籍，面向那些有志从事搜索引擎行业的青年学生、需要完整理解并优化搜索引擎的专业技术人员、搜索引擎的营销人员，以及网站的负责人等。本书是从事搜索引擎开发的工程技术人员难得的参考书，也可作为大中专院校相关专业的教学辅导书。

图书在版编目（CIP）数据

走进搜索引擎 / 潘雪峰，花贵春，梁斌编著. —2 版—北京：电子工业出版社，2011.5
ISBN 978-7-121-13104-2

Ⅰ. ①走… Ⅱ. ①潘… ②花… ③梁… Ⅲ. ①网络检索 Ⅳ. ①G354.4

中国版本图书馆 CIP 数据核字（2011）第 041946 号

策划编辑：孙学瑛
责任编辑：孙学瑛
印　　刷：
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：18.75　字数：400 千字
印　　次：2011 年 5 月第 1 次印刷
印　　数：4000 册　定价：49.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

作者序

本书第 1 版出版到现在已经 3 年了。在这段不长的时光里，搜索引擎技术有了进一步的发展。其中比较突出的是，随着数据规模进一步增大，为提升用户体验，搜索引擎性能进一步优化；在更广泛的用户参与下，增强了基于用户行为进行效果改进的能力。这也使得本书有了改版以适应这些重大变化的必要。

基于此，本书第 2 版增加了搜索引擎性能调优、搜索引擎日志分析，以及基于学习进行排序优化三方面的内容，希望能让读者跟上搜索技术的发展潮流，在这一领域的前沿真切地感受到它的勃勃生机。

当前，搜索技术已经不再局限于搜索引擎本身，它所建立的一套驾驭互联网级别海量数据的架构和理念正日益扩展到整个信息技术领域。而随着世界的日益信息化、数字化、网络化，这些理念的深远影响还会进一步显现。这又将是一次新的科技浪潮。

时光流逝，却有如轮回。信息技术产业，甚至整个科技界，正是在这样的浪潮更迭中不断进步。从 AT&T 的有线电话到 IBM 的大型机，到 Apple 的 PC 机，到 Intel 的 CPU，到 Motorola 的无线通信，到 Microsoft 的操作系统，到 Cisco 的路由器，到 Google 的搜索引擎，概莫能外。一次次浪潮，一个个产业巨擘，终将随自己的时代而去，但它们所带来的影响却将投射在人类文明的历史上，永不消逝。

至于搜索的浪潮究竟将持续多长时间，在整个 IT 史上留下怎样的一笔，只有时间才能告诉人们答案。此时此刻，置身其中，让我们打开书本，接受浪潮之巅的洗礼，走进搜索引擎。

关于本书作者

作者潘雪峰，毕业于中国科学院计算技术研究所，工学博士。研究兴趣包括多

媒体内容分析、机器学习和互联网数据挖掘，现从事搜索引擎领域相关工作。

作者花贵春，目前在清华大学信息科学与技术国家实验室攻读博士学位，研究兴趣包括机器学习及其在搜索领域的应用。

作者梁斌，目前在清华大学信息科学与技术国家实验室攻读博士学位，研究兴趣包括大规模数据处理、搜索引擎和软件工程等。

致谢

笔者首先要特别感谢他们的妻子，感谢她们在繁忙的工作和学习之余，包揽了家里家外大大小小的事务，还在笔者们有所懈怠的时候，从精神上给予莫大的支持和鼓励。正是她们无私的支持，才使本书得以面世。

感谢电子工业出版社计算机图书出版分社孙学瑛女士和邓彩屏女士，她们除了参与了此书的创作过程，还为笔者提供了有关图书市场的宝贵信息，使得本书更加面向读者，面向市场。

感谢本书参考文献的作者们、搜索引擎研究界的学者们，以及为此书提出宝贵技术意见的业界同行，正是你们杰出的成就和无私的帮助，才使得本书有了写作的基础和必要。

由于笔者水平有限，加之搜索领域的发展日新月异，书中不足及错误之处在所难免，敬请专家和读者给予批评指正。

潘雪峰、花贵春、梁斌

2011 年 2 月

目 录

第1章　引言

1.1　搜索引擎概述

1.2　搜索引擎的主要需求

1.3　搜索引擎的４大系统

1.1 搜索引擎概述

随着互联网的蓬勃发展，建立在互联网之上的各种应用也层出不穷，其中最为成功的莫过于万维网（WWW）。万维网被称为“网中之网”，是互联网上最受欢迎的服务之一。它运用超文本技术为人们访问信息资源提供了巨大的方便，但也以非线性组织的构建方式使人们在信息海洋中彷徨。奥地利的鲁施在 1994 年接触万维网，并在其作品《令人吃惊的万维网》（aMAZEingweb）中表达了对万维网的感受：它有那么可观的潜力，却又是经常使探索者丧失方向的迷宫。

时至今日，万维网迷宫般的复杂和魅力还在继续。因为它每天都在不断地产生、更新或消失各种各样的网页。其魅力依然，然而复杂不在。正是由于诞生了搜索引擎这样伟大的技术，万维网复杂的局面才被打破。搜索引擎成为带领人们走出迷宫的灯塔，帮助千百万的网民便捷地找到重要的信息。

WordNet 上对搜索引擎的解释是：一种用来在计算机网络，特别是在万维网上检索各种文件的计算机程序。从本质上讲，如果将搜索引擎的搜索结果看做一种动态网页，那么这种动态网页通过提交的检索关键词聚合了各种重要、有价值并与关键词相关的网页。因此，与其说搜索引擎是一个查询系统，不如说它是一个用户定义的信息聚合系统。通过用户输入的查询关键词，搜索引擎推测用户的查询意图，然后快速地返回相关的查询结果，供用户选择。

对搜索引擎的理解也经历了一个漫长的过程，从早期的目录式搜索，到今天的全文搜索，人们对搜索引擎的认识也在不断地加深。今天，公认的搜索引擎有如下 3 种服务方式。

1.1.1 目录式搜索引擎

在万维网出现早期，信息检索通常通过人工发现信息，依靠编辑人员的知识进行甄别，并在此基础上进行分类。用户可以在这个分类结构中浏览，这就是我们熟知的目录检索系统。这种搜索引擎最有名的是早期的雅虎（Yahoo），以及国内的搜狐（Sohu）。该类搜索引擎因为加入了人的智能，所以信息准确且导航质量较高。目录式搜索引擎的特点是检索的目标结果是网站，可以看做是网站的黄页查询；不

足是数据量有限、更新不及时，并且人工维护成本较高。

1.1.2　全文搜索引擎

全文搜索引擎是针对万维网所有网页进行全文检索的搜索引擎，由下载系统以某种策略自动地在万维网上搜集和发现信息，由索引系统为搜集到的信息建立索引，由查询系统根据用户的查询输入检索索引库，并将查询结果返回给用户。服务方式是面向网页的全文检索服务。该类搜索引擎的优点是信息量大、更新及时，并且无须人工干预；缺点是返回信息过多，有很多无关信息，用户必须从结果中筛选。其代表是谷歌（Google）及百度等第二代商用搜索引擎。

1.1.3　元搜索引擎（Meta-Search Engine）

这类搜索引擎没有自己的数据，而是将用户的查询请求同时向多个搜索引擎递交。然后将返回的结果进行重复排除及重新排序等处理后，作为自己的结果返回给用户。服务方式为面向网页的全文检索。这类搜索引擎的优点是返回结果的信息量大；缺点是不能够充分使用原搜索引擎的功能，用户需要做更多的筛选，其代表是WebCrawler。

上述3种搜索引擎共经历了不到20年的发展历程，然而就是在这短短的时间里，在一代代的搜索技术精英不断地努力下，成就了今天伟大而卓越的搜索引擎技术。其中的很多技术成果也用到了其他领域，创造了巨大的价值。

搜索引擎作为一个系统，它解决的是怎样的问题，主要设计目标是什么，主要分为哪几个部分？接下来我们在搜索引擎总体上进行一些讨论。

1.2　搜索引擎的主要需求

随着万维网上信息爆炸性地增长，传统的搜索方法无法为网民提供有效的搜索服务。万维网的发展迫切地要求一种快速、全面、准确、可靠且代价低廉的信息搜索方法，而具有全文检索的搜索引擎正满足了这 5 个需求，所以奠定了其在科学技术上的高度。有人甚至把搜索引擎和操作系统并列为当今最为复杂的系统软件，下面我们将对搜索引擎中最为主要的 5 个需求，同样也是搜索引擎的主要特点加以说明。

1.2.1 快

一方面，随着信息化社会的到来，信息可以说是无处不在，人们的日常生活离不开这些有价值的信息；另一方面，人们的生活节奏也在不断地加快，人们应该能够平等地获得这些公众信息。这就要求搜索引擎必须能够存储这些无处不在的信息，并且能够快速地进行信息搜索，满足网民的信息检索需求。

有调查表明，当今公开的搜索引擎的查询速度都在“秒”这个量级以下，商用搜索引擎的查询速度达到毫秒级，并且能够支持大规模用户的同时访问。

影响速度的原因很多，例如分词的效果、索引库的效率、分布查询的处理能力和查询缓存的命中率等，这些将在第 3 章和第 4 章中详细介绍。

1.2.2 全

在传统信息检索（Information retrieval）中，将查全率（Recall）作为衡量检索是否全面的度量指标（查全率也称作召回率），查全率是查询出的相关网页数和全部相关网页数的比率。例如在搜索引擎中查询“XML”，如果世界上包含“XML”这个关键词的网页数为 M，而实际该搜索引擎检索出这 M 条中的 N 条网页，那么查全率为 $N/M\times100\%$。

是否能查得全，主要取决于网页索引库的大小。如果网页库只包含了2 条XML的查询结果，即便都检索出来了，查全率也是极低的。可见，索引的网页数越多，越有助于提高查全率。

1.2.3 准

在传统信息检索中，应用查准率（Precision）作为衡量检索是否准确的指标，查准率是检索出的相关文档数与检索出的文档总数的比率。例如在搜索引擎中查询“XML”，在实际检索出的网页数 N 中，只有 P 个网页是与查询“XML”相关（Relavant）的，那么查准率为 $P/N\times100\%$。

通过图 1-1，可以全面理解查全率和查准率的关系。

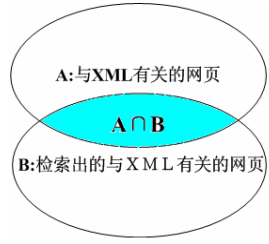


图 1-1　查全率和查准率的关系

查全率=$\frac{|A\cap B|}{|A|}$，其中对集合取| |运算的结果表示集合的数量。

查准率=$\frac{|A\cap B|}{|B|}$。

在搜索引擎这种特殊的检索实践中，查全率往往是不重要的。衡量的意义也不大，因为没有一个用户会把所有与查询相关的网页都浏览一遍。一般情况下，用户最为关注的仅仅为搜索结果中的前几条。而查准率在很大程度上决定了搜索的质量，在前10 条搜索结果（搜索结果首页）中满足用户的查询目的，这是搜索引擎查准率的主要体现。

是否能查得准，主要取决于网页排序。常见的有PageRank 等排序方法，在第 3 章中将介绍这方面的内容，在第 7 章中也会做详细介绍。

1.2.4　稳

毫无疑问，搜索引擎必须是一个能够长期并稳定地提供服务的系统，因此系统的稳定运行是很重要的需求。特别是商用搜索引擎，其稳定性被提高到了相当的高度。在任何情况下可以牺牲检索质量和检索速度，但必须能够提供持续的信息检索服务。

对于搜索引擎来说，查询来自四面八方，查询词也千差万别，同时进行的查询量也非常巨大。稳定地满足这些查询需要，需要在系统的结构上做出权衡，在文件存储方式、查询系统和索引系统设计等方面都需要考虑稳定性的因素。

1.2.5　省

由于搜索引擎处理数百亿的网页信息，同时每天接受来自数十亿用户的搜索请

求，搜索引擎的高能耗，已成为众矢之的。哈佛大学物理学者魏斯纳—葛洛斯研究指出，如果以台式计算机在 Google 网站执行两次搜索，所制造的二氧化碳量相当于煮一壶茶。

搜索引擎的耗费主要来自三个方面：电能，带宽，机器折旧。因此如果完成同样的工作尽可能用更少的机器，尽可能采用低能耗的机器，或者采用更低能耗的空调，这都可以大大节约能耗，在技术上，使用更少的机器是我们关注的话题，我们会在优化一章中做简要介绍。

1.3 搜索引擎的 4 大系统

搜索引擎在本书中被分为下载、分析、索引和查询 4 大系统进行论述。这 4 大系统相互配合，共同实现了搜索引擎的快、全、准、稳的 4 个主要需求，而本书最后的优化章节主要从“省”这个需求考虑，使得有限的资源可以发挥最大的效能。

1.3.1 搜索引擎的体系结构

搜索引擎的结构清晰，分工明确。按照各自的功能划分，分为以下 4 大系统：

（1）下载系统；

（2）分析系统；

（3）索引系统；

（4）查询系统。

其中下载系统负责从万维网上下载各种类型的网页，并且保持对万维网变化的同步，将在第 2 章中详细介绍。

分析系统负责抽取下载系统得到的网页数据，并进行 PageRank 和分词计算，将在第 3 章中详细介绍。

索引系统负责将分析系统处理后的网页对象索引入库，将在第 4 章中详细介绍。

查询系统负责分析用户提交的查询请求，然后从索引库中检索出相关网页并将

网页排序后，以查询结果的形式返回给用户，将在第 5 章中详细介绍。

搜索引擎简要结构如图 1-2 所示。

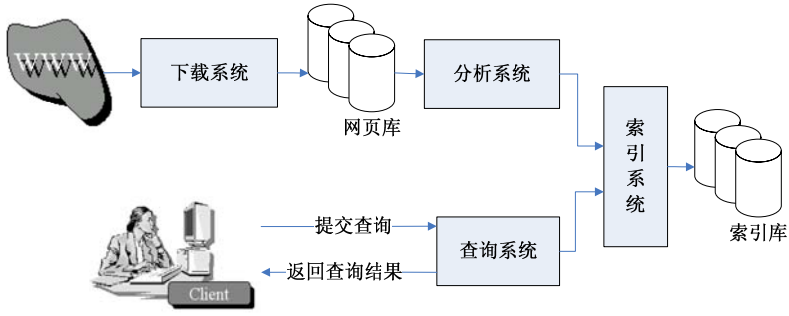


图 1-2　搜索引擎的简要结构图

从整体上看，下载系统、分析系统和索引系统组成了搜索引擎的数据制作部分，被称为是“离线部分”（offline part）；查询系统为搜索引擎的数据服务部分，要求快速响应，因此被称为“在线部分”（online part）。按照离线和在线划分，搜索引擎又可分为在线系统和离线系统。其中在线系统需要毫秒级的访问速度，而离线系统则没有时间性的严格限制，有些需要长达几周的时间才能计算完毕。

从细节上看，网页从开始到最后都是网页。而在搜索引擎的内部会有两种不同的形式，一种以网页库的方式存储；一种成为网页对象被存储在索引库中。搜索引擎的主要数据来自网页，网页处理能力是搜索引擎面对的主要挑战，下载系统和索引系统分别会进行一些关于数据存储的相关知识的介绍。

一个网页在万维网（WWW）中诞生，然后被下载系统下载，进而被分析并索引入库，最后因为该网页包含的一个关键词被检索而进入用户（Client）的大脑。这样一个奇妙的旅行都经历了哪些细节？各个系统内部如何工作？搜索引擎的全部画卷将在接下来的 4 章中按照这个顺序一一展开。

第 2 章　搜索引擎的下载系统

2.1　爬虫的发展历史

2.2　万维网及其网页分析

2.3　有关爬虫的基本概念

2.4　网页抓取原理

2.5　网页库

2.6　下载系统回顾及未来发展

2.1　爬虫的发展历史

在搜索引擎的 4 大系统中，第 1 个系统是下载系统。和航天运载火箭系统的动力系统一样，下载系统是搜索引擎大厦的基础。搜索的数据均来自于下载系统的工作，其工作方式巧妙、合理且强大。爬虫也称为“Crawler”，中文译为“爬虫”，或者“蜘蛛”。

2.1.1　世界上第 1 个爬虫

爬虫是一种自动抓取万维网网页信息的机器人。世界上第 1 个网络爬虫由麻省理工学院（MIT）的学生马休·格雷（Matthew Gray）在 1993 年写成，并命为“万维网漫游者”。尽管其编写目的不是为了做搜索引擎，但正是这革命性的创新，为搜索引擎的发展和今天的广泛应用奠定了坚实的基础。

2.1.2　爬虫的发展历程

现代搜索引擎的思路源于 Wanderer，不少人改进了 Matthew Grey 的蜘蛛程序。1994 年 7 月，Michael Mauldin 将 John Leavitt 的蜘蛛程序接入到其索引程序中，创建了当时著名的搜索引擎 Lycos（http://www.lycos.com/）。其后无数的搜索引擎促使爬虫越写越复杂，并逐渐向多策略、负载均衡及大规模增量抓取等方向发展。爬虫的工作成果使得搜索引擎能够检索几乎全部的万维网网页，甚至被删除的网页也可以通过一个称之为“网页快照”的功能访问。

前人的辉煌成就令人赞叹不已，那么爬虫是怎么实现这些功能的呢？为什么说它巧妙、合理且强大呢？让我们首先从爬虫开始入手，深入理解搜索引擎的下载系统。

2.2　万维网及其网页分析

在深入爬虫领地之前，不妨把爬虫看做是劳动者，把机器和网络带宽资源看做是劳动资料，把万维网上的网页看做是劳动对象。因此，深入并透彻地理解劳动对

象才能更加科学地理解劳动者的工作原理。

2.2.1 蝴蝶结型的万维网

首先从万维网的静态结构入手，如果将万维网定义为一个相互连通的连通图，网页为结点，链接（link）为边，那么任意一个网页可能被其他网页链接，这种链接称为“反向链接”（back link）。这个网页也可能链接到其他网页，这种链接称为“正向链接”（简称“链接”）。

Broder 通过一个有趣的 Random-start BFS（Breadth-First Search）实验随机取得 570 个网页作为开始结点，依次逐个地试验这 570 个网页。首先采用正向（forward direction）宽度优先遍历（宽度优先遍历的方法将在后面检索），即按照网页到网页的链接进行遍历。然后采用反向（backward）宽度优先遍历，即按照网页到网页反向链接进行遍历。例如网页 A 包含网页 B 的链接，则从网页 A 开始，正向遍历可以遍历到网页 B；从网页 B 开始，反向遍历可以遍历到网页 A。

通过实验发现，无论是正向遍历还是反向遍历，表现出来的是截然不同的遍历效果。要么是在遍历到很小的结果集（90%的情况下探测到的集合大小不超过 90 个结点，极端情况下也只有几万个网页而已）后遍历终止；要么是爆炸性地遍历到了大约 1 亿个网页，但是没有探测到全部的 18 600 万个结点。此外，对于一部分的起始点，无论是正向，还是反向遍历都能够探测出大约 1 亿个网页，这部分起始点属于后面提到的 SCC 部分。通过实验数据，Broder 得出这样一个结论，即万维网具有蝴蝶结型（bow tie）结构，如图 2-1 所示。

其中的网页分为如下 4 种类型，各约占 1/4。

（1）蝴蝶结的中部（SCC，Strongly Connected Component）。

这种类型的网页彼此相连，任意去掉有限个网页，不会影响其连通度。Random-start BFS 遍历实验的起始点选择该部分网页，采用正向遍历的方法，从统计的角度上看可以遍历占全部网页 3/4 的网页数；采用反向遍历的方法也可以遍历大致同样数量的网页。

（2）蝴蝶结的左部（IN）。

这种类型的网页指向中心部分（SCC），称为“目录型网页”（hub page），即通

常说的导航网页。Random-start BFS 遍历实验的起始点选择该部分网页，采用正向遍历的方法可以遍历占全部网页 3/4 的网页数；采用反向遍历的方法只能遍历很有限的一些网页，所占比例可以忽略不计。

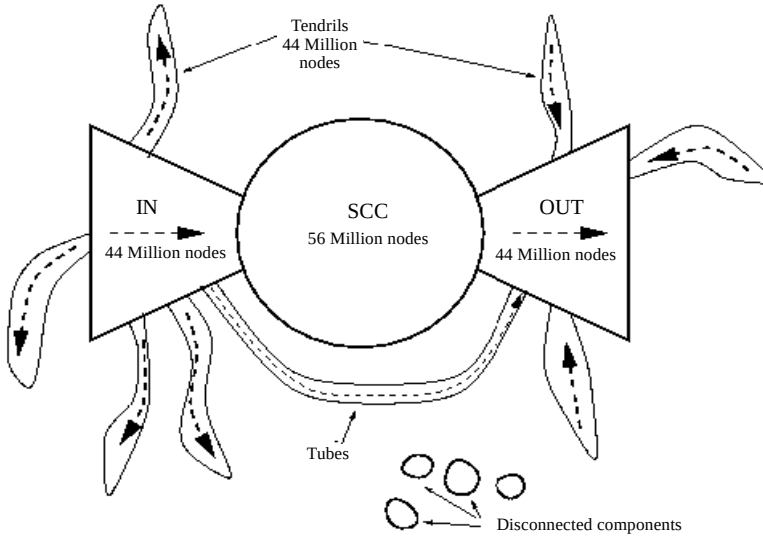


图 2-1　万维网的蝴蝶结型结构

（3）蝴蝶结的右部（OUT）。

这种类型的网页被中心部分指向，称为“权威性网页”（authority page）。这些网页被引用次数多，表示为大多数网页对其“认可度”高。Random-start BFS 遍历实验的起始点选择该部分网页，采用正向遍历的方法只能遍历有限的网页；采用反向遍历的方法可以遍历占全部网页 3/4 的网页。

（4）蝴蝶结的须脚（Tendrils）。

这种类型的网页表现为从左部链出到其他网页，或者其他网页链入右部或从左部直接链入右部，以及少部分与中部、左部或右部都没有链接（非连通分量 DISC）。Random-start BFS 遍历实验的起始点选择这部分的网页，无论采用何种遍历方法都只能遍历有限的网页。

通过万维网的结构特征得出如下两个结论。

（1）爬虫尽可能选择蝴蝶结的左部，或者中部的网页为起始访问结点集合

（starting set of URLs）进行遍历，这样可以得到尽可能完整的遍历效果。如果选择右部或者须脚部分的网页为起始结点，则只能抓取很有限的网页。

（2）网页分为目录型网页和权威性网页，目录型网页为普通网民服务，便于网民点击从而继续浏览更多的网页。该部分网页对于深入抓取权威性网页有重大意义；权威性网页是那些处于蝴蝶结中部或者右部的网页，这类网页的反向链接数很多，而正向链接数相对较少，通常认为这类网页比较重要。

2.2.2 万维网的直径

万维网直径（Diameter of the World Wide Web）又称为“Web 直径”，其定义为如果用 d 表示存在一条从网页 u 到网页 v 的路径，那么这些万维网上所有不同的连通网页对（pair of connected page）所构成的最短路径的平均长度即 Web 直径。[Reka Albert et al.1999]一文中通过实验测定，拟合出计算万维网的经验公式：$d = 0.35 + 2.06 \log(N)$，其中论文中 N 取 $8*10^8$，当时得到的结论是互联网直径约为 19。近些年来，对该问题没有进一步的研究，在网页数量爆炸的今天，虚假、重复网页和链接泛滥，互联网直径的测定存在很大困难，学术上一般沿用了这个结果。

有研究表明，网页的平均出度为 25.7，即平均每一个网页含有 25.7 个指向其他网页的链接。这些链接引导网民继续在网上冲浪，直到对某个主题厌倦，开始下一个主题。下一章中还会讲到这个冲浪模型。

同样得到如下两个结论。

（1）遍历的方法很大程度上影响了爬虫的效率，万维网的网页结构并没有我们想象的深，却出乎我们意料的宽，因此爬虫的遍历方式多采用宽度优先的遍历方式。当然这里还有一个网页重要性的原因，采用这种方式可以较好地抓取重要性高的网页。

（2）万维网错综复杂，任选一个抓取路线不能保证总是最优。为了防止爬虫一路走到黑，充分考虑万维网的万维网直径后，采用“深度策略”控制抓取深度，从而完美地解决了这个问题。

2.2.3 万维网的规模及变化特征

在了解了万维网宏观静态的一面之后，再来看看其宏观动态变化的一面。

毫无疑问，万维网的规模是巨大的。早在 1998 年，各种研究就在试图估计万维网的规模。尽管方法各异，得出的具体数值不尽相同，但都认为万维网的规模（有价值的网页数目）在十亿数量级上。根据 Lawrence 和 Giles 在 1999 年的研究显示，世界万维网网页的数目每两年增加一倍，因此截至目前万维网的规模已达百亿数量级。

2000 年斯坦福大学的 Cho 和 Garcia-Molina 随机选用 50 万个网页做样本，发现 23%的网页是每天更新的，其中 40%域名后缀为.com 的网页是每天更新的，网页的半衰期（half-life）为 10 天。也就是说 50 万的网页在 10 天后只剩下 25 万，再过 10 天只剩下 12.5 万。此外，网页的变化可以归结为泊松过程（Poisson process）模型（在网页重访策略部分会详细提到这个模型）。可以说万维网的变化是日新月异、一日千里，如何与万维网的变化保持同步，成为爬虫的又一个主要难点。

2.2.4　网页的特征

让我们从宏观来到微观，考察一下网页的特性。归纳起来，对于搜索引擎来说，网页具有的 3 大特征分别是挥发性（volatile）、半结构性（semi-structured）和隐蔽性（hidden）。网页诞生后到网页的消亡（挥发性）总是不可避免的，HTML 语言描述的网页是一种半结构化的数据。除了静态网页以外，还有很多隐藏的动态网页，例如需要登录才能看到的某些动态网页。

接下来从有关爬虫的基本概念开始，由点到面，层层深入地理解爬虫是如何针对万维网的这些特点，采取相应的策略实现了抓得全、抓得快且代价低的基本目标。

2.3　有关爬虫的基本概念

2.3.1　爬虫

爬虫也称为“Wanderers”（漫步者）或者“Robots”（机器人），它首先是一组运行在计算机中的程序，在搜索引擎系统中负责抓取时新的且公共可访问的 Web 网页、图片和文档等资源。这种抓取的过程为通过下载一个网页，分析其中的链接，继而漫游到其他链接指向的网页，循环往复。

2.3.2 种子站点

种子站点是爬虫开始抓取的起点，通常为各大门户网站和官方网站的首页等。

2.3.3 URL

URL 是“Uniform Resource Locator”（统一资源定位器）的缩写，它是用在万维网和其他万维网资源中的一种编址系统，例如 http://www. nju.edu.cn/index.html，其中包含访问方式（http 协议）、被访问的服务器（www.nju.edu.cn）和被访问的文件（index.html）。

2.3.4 Backlinks

一个网页的 Backlinks 是那些除网页自身之外指向自身链接的集合，Backlinks 的数目是衡量网页受“欢迎”程度的重要度量方式。

2.4 网页抓取原理

2.4.1 telnet 和 wget

使用 Windows 操作系统的用户，执行如下步骤即可下载一个网页。

（1）打开 Windows 命令行窗口。

（2）输入 telnet www.nju.edu.cn 80（注意中间的空格）。

（3）输入 GET /index.html（注意 GET 要全大写，其后要有空格）。

如果以上每一步都正确执行，则应该在桌面上看到南京大学网站首页的网页源代码。笔者得到的是如下的网页代码：

```
<script>location.href="./cps/site/NJU/nju/nju.htm"</script>
```

使用 Linux 操作系统的用户，则只需要一步，即输入：vim http://www.nju.

edu.cn/index.html，可以得到同样的结果。

如果要把该网页文件下载到本地硬盘，对于 Linux 操作系统的用户，只需要输入命令：

```
wget www.nju.edu.cn/index.html
```

之后使用 vi 就可以打开该文件。Windows 操作系统的用户可以下载一个 wget 程序，使用同样方法下载网页。

由此看来，下载一个网页如此简单，如果要下载整个万维网，那么应当采用什么样的遍历规则呢？

2.4.2　从种子站点开始逐层抓取

基于万维网的蝴蝶结型结构，这种非线性的网页组织结构，就出现了抓取“顺序”的问题，即哪些先抓、哪些后抓。这种解决抓取“顺序”的策略必须保证尽可能抓取所有网页（本章不区分抓取网页和下载网页的区别）。

一般来说，爬虫选择蝴蝶结左部的网页。即目录型网页作为种子站点（抓取出发点），典型的如 sina.com 和 sohu.com 这样的门户网站的主页。每完成一次抓取网页之后提取其中的链接（提取的方法需要一些 HTML 语法分析，以及区分绝对路径和相对路径的技巧等），这些字符串形式的链接是指向其他网页的 URL，它们指引爬虫更加深入地抓取其他网页。一个网页常常包含多个链接，因此在提取网页的链接后，如何继续抓取其他网页，爬虫有如下两种选择处理抓取的“顺序”问题。

（1）深度优先策略（Depth-First Traversal）。

深度优先的遍历策略类似中华文化中家族继承的策略，典型的如封建帝位的继承。通常为长子继承，如果长子过世，长孙的优先级大于次子的优先级。如长子过世且长子无子，那么次子继承，这种在继承上的优先关系也称作深度优先策略，如图 2-2 所示。

在继承优先顺序上，长子>长孙>长孙的其他兄弟>次子>次子的其他兄弟。这种首先选择某个分支，继而深入到不能深入的情况下才考虑其他分支的策略即为深度优先策略。

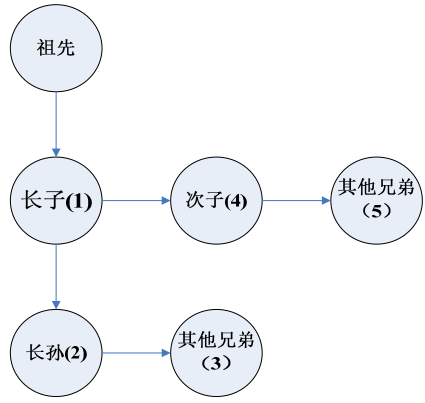


图 2-2　深度优先策略

（2）宽度优先策略（Breadth-First Traversal）。

宽度优先也称为“广度优先”，或“层次优先”，它是一种层次型距离不断增大的遍历方式，类似长幼有序的规则。在晚辈给长辈献茶时，总是先献长辈，然后次之，如图 2-3 所示。

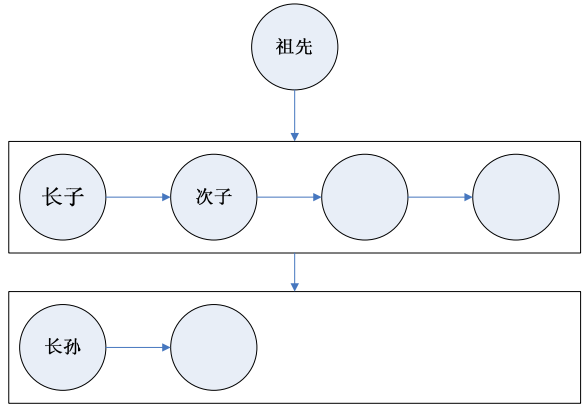


图 2-3　宽度优先策略

在图 2-3 中，祖先的优先级最高，第 2 层的优先级大于第 3 层，每层的内部优先级以年长者优先。因此这里次子的优先级大于长孙的优先级，这就是宽度优先策略。

在抓取的顺序策略上选择宽度优先出于如下 3 点原因。

首先，重要的网页往往离种子站点的距离较近，这符合直觉。我们通常在打开

某些新闻网站时，进入眼帘的往往是最重要的新闻。随着不断地冲浪（可以理解为深度不断加深），所看到的网页的重要性越来越低，甚至偶尔会出现无法访问的情况。

其次，万维网的深度没有我们想象得那么深，到达某一个网页的路径通常很多，总会存在一条很短的路径到达。有研究表明，中文万维网直径的长度只有 17。

最后，宽度优先规则有利于多爬虫合作抓取（这种合作策略在后面还会提到）。这是因为该规则开始抓取的网页通常都是站内网页，逐渐才会遇到站外链接，因此抓取的封闭性较强。

进行宽度优先遍历时，必须要有一个队列（queue）数据结构支持。这个队列理解为其工作负载队列，只要其中存在没有完成的抓取任务，就需要提取队头位置的网页继续抓取。直到完成全部抓取任务，工作负载队列为空为止。详细的抓取的过程如图 2-4 所示。

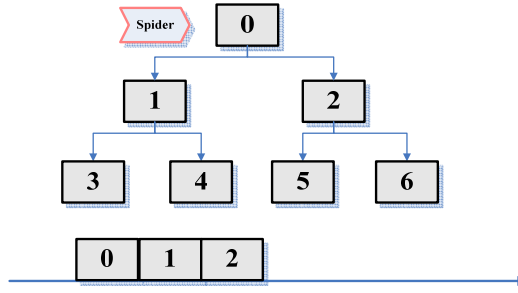


图 2-4　详细的抓取过程

（1）爬虫提取工作负载队列中的网页 0（这里假定网页 0 是种子站点，即抓取的起始结点），抓取后首先对网页 0 进行链接分析，将提取的网页 1 和网页 2 的链接放到待抓取工作队列中。此时当前工作负载队列中增加了网页 1 和网页 2，如图 2-5 所示。

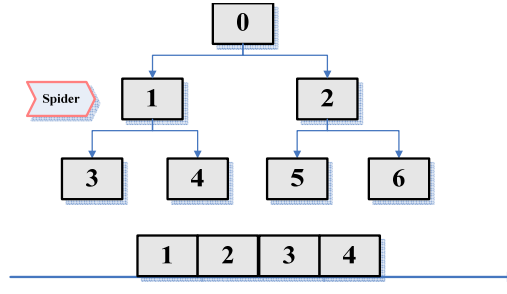


图 2-5　工作负载队列中增加了网页 1 和网页 2

（2）工作负载队列中的网页 0 抓取完毕，继续抓取工作队列中第 1 个任务，即网页 1，并提取指向网页 3 和网页 4 的链接放到工作队列中，如图 2-6 所示。

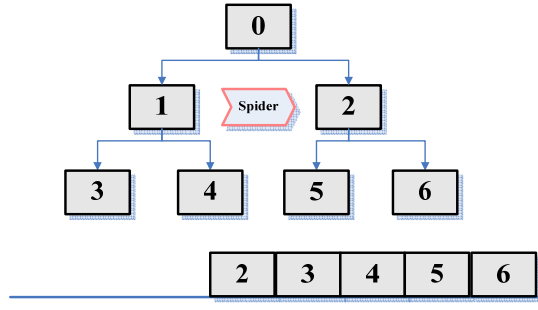


图 2-6　网页 3 和网页 4 的链接放到工作队列中

（3）根据宽度优先的规则抓取工作队列中的网页 2（注意，这里可以理解为次子的优先级大于长孙，即网页 2 要在网页 3 之前抓取），并且分析出指向网页 5 和网页 6 的链接放入工作队列中，如图 2-7 所示。

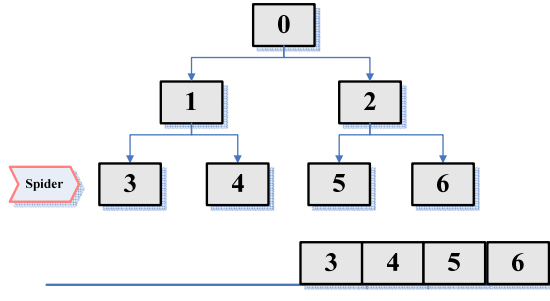


图 2-7　网页 5 和网页 6 的链接放入工作队列中

（4）继续抓取工作队列中的网页 3，直到结束。

如果将余下的步骤略去，这种按宽度优先顺序的抓取序列就展示在我们的眼前了，即网页 0~6。

通过这个例子回顾一下采用宽度优先遍历的策略抓取网页带来的好处。

如果将网页 0 理解为门户首页，那么距离门户首页越近的网页越重要。把首页理解成一个窗口，那么打开这个窗口，距离越近的网页被浏览的机会越大，因此也就越重要。上面这个例子中，网页 1 比网页 3 和网页 5 更加重要，按照距离递增的宽度优先抓取顺序恰好符合了重要网页优先抓取的要求。如果按照深度优先规则抓

取，不仅破坏了重要优先的原则，而且破坏了抓取的封闭性。从而不利于多爬虫的合作抓取，在合作抓取策略中还将提到抓取封闭性问题。

然而，以上的例子是不完备的。为了简化描述，这里使用树型结构。树的一个基本特性是任何两点之间只有唯一的路径，即不会出现环路。而万维网中存在大量环路，实际上网页 6 可能存在指向网页 0 的链接。如果没有任何判断，则爬虫永远抓取不完，因为存在了 0、2 和 6 这样一个死循环。死循环会带来下面两个不利后果。

（1）不该抓的反复抓。如果一个网页变化不大，则只要抓取一次即可，重复抓取相同网页就会占用大量 CPU 和带宽资源。

（2）该抓的没有机会抓。由于大量的资源被用在做无用功，所以必然使得某些网页被歧视，从而不能被及时抓取。

为了解决死循环的问题，通常需要一个称为“不重复抓取策略”和一个称为“深度策略”的方法来解决。

2.4.3　不重复抓取策略

不重复的关键在于记住历史，只有记住过去才可能不重复。那些不需要记忆历史的程序设计往往都比较简单，从计算机的角度上讲即有限状态机模型。每个状态都是对历史的积累，而没有记忆历史，这种只需要有限的存储即可完成的工作通常都比较简单。而那些需要记住历史的程序相对复杂，例如迷宫求解程序，则需要一个栈（stack）结构记住走过的地方。从而在失败时可以回溯，继续寻找出路。而通过动态规划寻找问题最优解等程序，也都需要一个记住历史的功能才能保证不重复计算。

爬虫记录历史的方式是哈希表（也称为“杂凑表”），每一条记录是否被抓取的信息存放在哈希表的某一个槽位上。如果某网页在过去的某个时刻已经被抓取，则将其对应的槽位的值置 1，反之置 0。而具体映射到哪一个槽位，则由哈希函数决定。在介绍哈希表前，我们首先简单了解一下 MD5 签名函数。

1992 年 8 月，Ronald L. Rivest 在向 IEFT 提交的一份重要文件中描述了这种 MD5 签名算法的原理。由于这种算法的公开性和安全性，所以在 20 世纪 90 年代被广泛使用。MD5 签名是一个哈希函数，可以将任意长度的数据流转换为一个固定长度的数字（通常为 4 个整型，即 128 位）。这个数字称为“数据流的签名”或者“指

纹”（Digital Finger Print），并且数据流中的任意一个微小的变化都会导致签名值发生变化。

将 URL 字符串数字化是通过某种计算将任何一个 URL 字符串唯一地计算成一个整数，如图 2-8 所示。

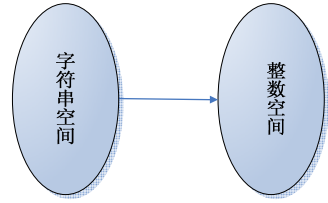


图 2-8　字符串数字化

在一个 URL 哈希函数映射下，任意一个字符串都唯一地对应一个整数。一个 64 位整数可以表达 1 800 万 TB（1TB = 1 000 GB），而字符串空间的大小远远大于 64 位整数所表达的整数空间大小，因此碰撞是不可避免的。碰撞是指那些字符串不同，而计算出相同的签名值的情况。如果杂凑函数设计得足够好，则相互碰撞的机会可以小到忽略不计。

更加理论化的表示如下：

令 $\boldsymbol{U}$=128 位整数集合，$\boldsymbol{S}$ =字符串集合：

$$\forall \boldsymbol{URLi} \in \boldsymbol{S}, \boldsymbol{URLj} \in \boldsymbol{S}$$
$$Ti = MD5(\boldsymbol{URLi})$$
$$Tj = MD5(\boldsymbol{URLj})$$

其中 $Ti \in \boldsymbol{U}, Tj \in \boldsymbol{U}$ ， $\boldsymbol{URLi} \neq \boldsymbol{URLj}$ ，则 $P(Ti = Tj) < \varepsilon$ 。

不难看出，对于两个不相同的 URL，即 $\boldsymbol{URLi}$ 和 $\boldsymbol{URLj}$ 通过哈希函数分别计算出各自的签名值 Ti 及 Tj，这两个签名值相等（碰撞）的概率小于一个足够小的小整数 ε 。正是由于 MD5 函数符合这种特性，因此被广泛地用做哈希函数。

标准 MD5 签名的整数空间很大，128 位整数能表达 2 的 128 次方个不同的数，这是十分巨大的。而实际分配的哈希表空间却十分有限，一个普通 32 位处理器，理论上最多可以分配 2 的 32 次方大小的内存，即 4G 大小的内存（Linux 系统中操作系统内核还需占用 1G 内存，实际上可供搜索引擎使用的极限内存只有 3G）。

因此，在实际处理中将签名值进行模运算映射到实际的哈希表中（可以理解为哈希表存放在内存中）。哈希函数可以是 MD5（URL）%*n*（%表示取模运算），这样使得一个 URL 被映射到大小为 *n* 的哈希表的某个槽位上。

哈希表是简单的顺序表，即数组。从实际应用的角度看，这个数组要足够大，而且尽可能地全部放在内存中，以保证每一个 URL 的签名都可以通过查找哈希表来确定是否曾经抓取过，其示例如图 2-9 所示。

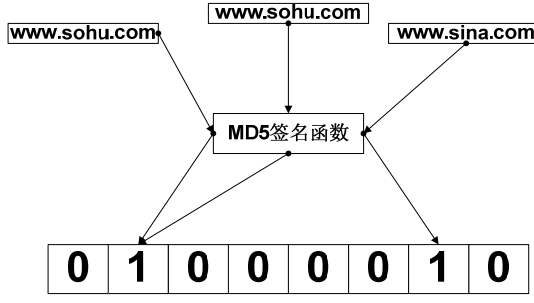


图 2-9　哈希函数示例

假定抓取顺序为 www.sohu.com、www.sina.com 和 www.sohu.com。首先我们在抓取 www.sohu.com 时为该 URL 签名，假定得到的结果为 2，我们存放在数组的第 2 个槽上。这个过程只要将第 2 个槽位置 1，表示已访问过。继续抓取 www.sina.com，以同样方法，假定将其存放在数组的第 7 个槽位上。之后重复抓取到了 www.sohu.com。由于 MD5 签名函数特性，任何一个自变量唯一对应一个应变量，因此计算 www.sohu.com 的签名，同样得到结果为 2。当我们查询第 2 个槽位时，发现该槽位已经置 1，说明已经访问过，从而不再重复访问。

有调查表明，中文万维网页总数超过 100 亿。如果要存放 100 亿网页是否被抓取过这样一个历史信息，那么需要的哈希表将会非常大。如果哈希表每一个单元为一个字节（8 位），则至少需要 10 GB 的容量。而且为了保证碰撞的概率极低，实际需要的容量远大于 10 GB。然而 32 位操作系统的最大寻址空间为 4 GB，即理论上最大的物理内存为 4 GB，因此在内存中不可能分配一个如此巨大的哈希表。

为了能够将整个哈希表装入内存，可以采用 Bitmap 的数据结构压缩内存用量，Bitmap 需要借助按位与（&）和按位或（|），以及左移（<<），右移（>>）这 4 种基本位运算。通过一个例子来说明，如图 2-10 所示。

Bitmap 结构使用整型作为基本单元，一个整型为 32 位，一个 int Hash[8]的数组

仅通过 8 个单元来记录历史。如果将比特作为最小单元，则能扩展到 256 个单元。

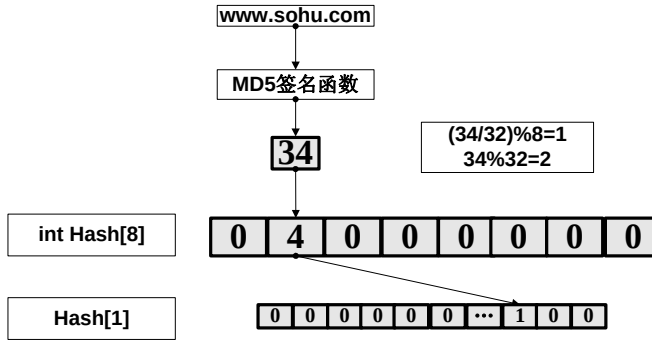


图 2-10　Bitmap 结构的哈希表

定义这样一个哈希表，即 int Hash[8]。将“www.sohu.com”作为字符串执行一次 MD5 签名，假定得到的签名值为 34。用 34 除以 32，得到商数为 1，继续用 1 对 8 取模运算得到 1，表示槽位在 Hash[1]中。用 34 除以 32，得到余数为 2，表示 34 映射到 Hash[1]这个 32 位整数的第 3 个比特位上（从低位算起，余数为 0 则映射到第 1 个比特位），将这个比特位置 1，则 Hash[1]为 4，这是因为 Hash[1]的第 3 个比特位被置 1。

不妨将 34 这个数按比特位的方式展开，如图 2-11 所示。通过这种形式进一步理解 Bitmap 如何扩展存储空间的方法。

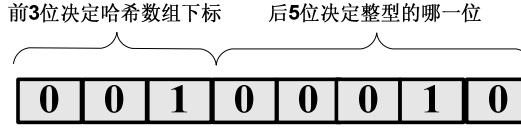


图 2-11　将 34 位按比特位表达

34 的二进制的前 3 位 001 决定了它存放在数组的第 2 个位置上（数组的起始下标为 0），即存放在 Hash[1]上，前面提到的除以 32 后，对 8 取模的目的就是取出 34 这个数的前 3 位。

具体存放在 Hash[1]的从低位算起的第几个比特位则由后 5 位决定。后 5 位为 00010，表示存放在 Hash[1]这个整数的第 3 个比特位上。前面提到的 34 除以 32 后得到余数即为这里的后 5 位。

通过前面的计算，Hash[1]的实际值为 4，表示其第 3 个比特位被置 1。图 2-11

中，在 Hash[1]中用 4 这个数值完整地表示了 34 这个签名值。

通常使用移位运算来快速计算除以或者乘以一个 2 的 *n* 次幂；将某个比特位置为 1，采用按位或运算；查询某个比特位是否为 1，采用按位与运算。

对前面计算的过程用 C 代码的方式表示如下：

（1）int MD5 = 34；

（2）int index_int = MD5&31；　// 相当于 34%32

（3）int index_Hash =(MD5>>5)&7；//相当于(34/32)%8，32 是 2 的 5 次幂

（4）if(Hash[index_Hash]&(1<<index_int)) //表示探测该槽位是否已经被置位，注意这里的按位与运算，查询某个比特位是否为 1 的技巧，1<<index_int 表示将 1 右移 index_int 位。

（5）else{ Hash[index_Hash]= Hash[index_ Hash] |(1<<index_int)；//表示该槽位没有被置位，用或运算置位。

上述过程，读者可以用纸笔计算一下，从而了解这种解决问题的技巧。通过 Bitmap 结构，利用计算机速度极快的按位与、按位或和移位运算能够高效地实现记住历史，不重复抓取的策略，同时将原来的哈希表压缩了 1/32。

除此之外，Bloom filter 算法还可以更加经济地利用好哈希表中的比特位，使得更加经济地灵活地使用内存。简单地说，对于一个字符串独立地使用 *m* 哈希函数计算，然后将其值映射到哈希表的 *m* 槽上。如果这 *m* 个槽都置 1，说明该字符串已经在历史上出现过；否则说明该字符串是第 1 次出现，将 *m* 个槽都置 1。通过图 2-12 来进一步理解。

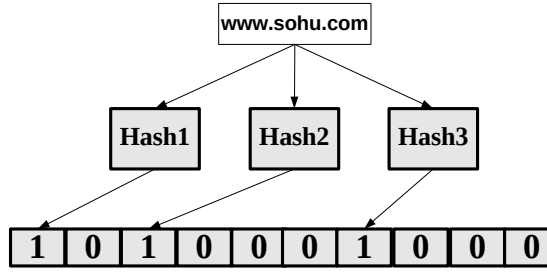


图 2-12　Bloom filter 原理

在抓取到 www.sohu.com 时，执行 3 次独立的不同的哈希函数（如图 2-12 中的 Hash1、Hash2 和 Hash3）运算。映射到哈希表的 3 个槽上，假定是 1、3 和 7。接下来如果抓取到 www.sina.com。经过同样的 3 次哈希函数运算，假定计算的结果是 1、3 和 6。很明显存在一个槽 6 是之前没有置位的，因此 www.sina.com 没有抓取过。成功抓取 www.sina.com 后，将槽位 6 置 1。Bloom 算法一方面提高了哈希数组的利用效率，并被证明适宜于应用在大规模集合查找计算中，同时在网络应用中也被大量采用。有兴趣的读者可以进一步阅读文献 [Bloom，B.H. 1970]深入理解该算法。

在抓取提速部分讲到通过多爬虫的合作抓取，这样可以进一步降低每个爬虫的用于记录历史抓取情况的哈希表大小。即如果有 *N* 个爬虫，则可将哈希表继续压缩到原有大小的 *N* 分之一。如果 *N* 个爬虫分别运行在不同的机器上，那么每个机器被哈希表占用的内存用量将非常少，通常保持抓取历史记录所需要的内存在百兆字节左右是恰当的。

通过不重复抓取的方法初步解决了死循环的问题，即抓过的不再抓，循环的条件被破坏。然而实际操作中还有这样一个隐含的问题，即如前所述的万维网直径。如果任意两个网页存在链接，则链接它们的最短路径的期望值为 17。这样，爬虫无论用何种遍历方法都不能保证一定会按照最佳路径抓取每一个网页，因为任何一个网页都可能从多个种子站点开始宽度优先的被遍历到。

我们来看一个例子。

假如从种子站点 A、种子站点 B 及种子站点 C 开始均存在一条到达网页 P 的路径，路径长度分别为 3、27 和 143。很明显，需要浪费两次检查网页 P 是否被抓取的开销。由于从种子站点 A 开始很快抓到了网页 P，而种子站点 B 和 C 则经过漫长的路径达到 P 时共计需要两次检查网页 P 是否被抓取，这显然是不够经济的。

为了防止爬虫无限制的宽度优先抓取，必须在某个深度上进行限制，到达这个深度后停止抓取，这个深度的取值就是万维网直径长度。当在最大深度上停止时，那些深度过大的未抓网页，总是期望可以从其他种子站点更加经济地到达。例如，种子站点 B 和 C 在抓取到深度 17 时即停止抓取，把抓取网页 P 的机会留给从种子站点 A 出发的进行抓取工作的爬虫。不难看出，限制抓取深度也破坏了死循环的条件，即使出现循环也会在有限次后停止。此外，深度策略和宽度优先遍历策略的组合可以有效地保证抓取过程中的封闭性，即在抓取过程（遍历路径）中总是在抓取相同域名下的网页，而很少出现其他域名下的网页。

爬虫不停地抓取网页，其工作负荷能力总是有限的，同时下载带宽和时间也是有限的，因此必须能够优先下载重要性高的网页。下面的网页抓取顺序策略保证了即使不能下载全部网页，也要保证能下载全部重要性高的网页。

2.4.4　网页抓取优先策略

网页抓取优先策略也称为“页面选择问题”（Page Selection），通常是尽可能地首先抓取重要性的网页，这样保证在有限的资源内尽可能地照顾到那些重要性高的网页。哪些网页才是重要性高的呢？如何量化重要性呢？

重要性度量由链接欢迎度、链接重要度和平均链接深度这 3 个方面决定[Arvind Arasu et al. 2001]。

定义链接欢迎度为 $IB(P)$，它主要由反向链接（Backlinks）的数目和质量决定。首先考察数目，直观地讲，一个网页有越多的链接指向它（反向链接数多），那么表示其他网页对其的认可，同时这个网页被网民访问的机会就大，推测出其重要性也就越高；其次考察质量，如果被越多重要性高的网页指向，那么其重要性也就越高。如果不考虑质量，就会出现局部最优，而不是全局最优的问题。最典型的就是作弊网页，人为地在一些网页中设置了大量反向链接指向其自身的网页，以提高该网页的重要性。如果不考虑链接质量，就会被这些作弊者所利用。

定义链接重要度为 $IL(P)$，它是一个关于 URL 字符串的函数，仅仅考察字符串本身。链接重要度主要通过一些模式，比如认为包含“.com”或者“home”的 URL 重要度高，以及具有较少斜杠（slash）的 URL 重要度高等。

定义平均链接深度为 $ID(P)$，此为笔者所创。$ID(P)$表示在一个种子站点集合中，每个种子站点如果存在一条链路（宽度优先遍历规则）到达该网页，那么平均链接深度就是这个网页的又一个重要性指标。距离种子站点越近，说明被访问的机会越多，因此重要性越高，可以认为种子站点是那些重要性最高的网页。离种子站点越远，重要性越低。事实上，按照宽度优先的遍历规则即可满足这种重要性高的网页被优先抓取的需要。

最后，定义网页重要性的度量为 $I(P)$，它由以上两个量化值线性决定，即：

$$I(P) = \alpha * IB(p) + \beta * IL(p)$$

平均链接深度由宽度优先的遍历规则保证，因此不作为重要性评价的指标。在

抓取能力有限的情况下，如果能够把重要性高的网页尽可能地抓完，是合理、科学的，最终被用户查询到的网页也往往是那些重要性高的网页。

尽管这样看来已经足够完美，事实上，还是忽视了一个重要的要素——时间。时间导致万维网动态变化的一面。如何抓取那些新增的网页呢？如何重访（revisit）那些被修改了的网页呢？如何发现那些被删除了的网页呢？为了保持和万维网网页的同步变化，就必须有网页重访策略。通过该策略可以识别增加、修改及删除网页这 3 种网页变化的情况。

2.4.5 网页重访策略

爬虫日积月累地下载各种各样的网页，然而过去的网页可能变化了。因此爬虫不得不周期性地刷新（refresh），重访那些已经下载的网页。通过重访以使得这些网页能够与万维网的变化与时俱进（up-to-date）。

文献[CHO, J. AND GARCIA-MOLINA, H. 2000a]表明网页的变化可以归结为泊松过程（Poisson process）模型。

为了描述泊松过程模型，使用 $X(t)$表示在一个$(0,t)$的时间段内网页发生变化的数目，一个参数为 λ 的泊松分布满足以下性质。

对于 $s \geqslant 0$，$t \geqslant 0$，随机变量 $X(s+t)-X(s)$符合泊松分布，即：

$$Pr\{\mathrm{X}(s+t)\text{-}X(s)=k\ \}=\frac{(\lambda t)^k}{k!}e^{-\lambda t}$$

其中 k=1,2,3…。

随机变量 $X(s+t)-X(s)$的期望值为 λt：

$$E[X(s+t)\text{-}\mathrm{X}(s)]=\lambda t$$

可以通过简单的方法验证，假定时间周期（时间间隔）为 1(t=1)，则有：

$$E[X(t+1)\text{-}X(t)]=\sum_{k=0}^{\infty}kPr\{X(t+1)-X(t)\}=\sum_{k=1}^{\infty}k\frac{\lambda^k e^{-\lambda}}{k!}=\lambda$$

其中：$\sum_{k=1}^{\infty}\frac{\lambda^{k-1}}{(k-1)!}=e^{\lambda}$

采用 $E[X(s+t)-X(s)]=\lambda t$ 的计算公式同样也得到了

$$E[X(t+1)-X(t)]=\lambda t=\lambda$$

Cho 和 Garcia-Molina 通过对 50 万个随机网页的跟踪分析，得出大部分网页的更新属于泊松分布的重要结论，如图 2-13 所示。

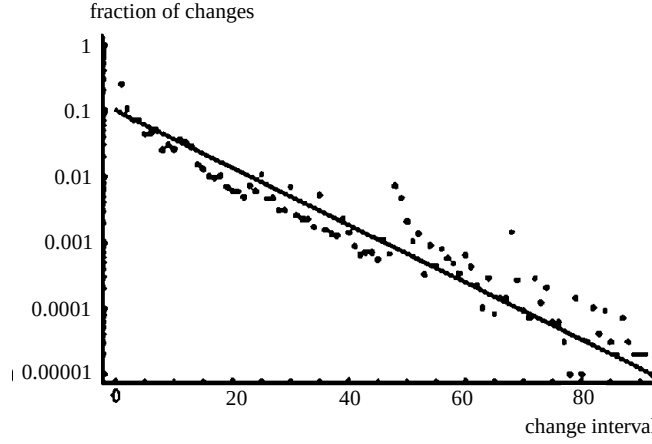


图 2-13　网页更新的泊松分布[CHO et al.2000a]

横坐标表示网页更新的时间间隔，纵坐标是在时间间隔下发生变化的网页所占比例（该比例取对数）。

在泊松模型下，若 t 是泊松过程中下一个事件发生的间隔时间，则下一个事件发生的时间间隔的概率分布符合指数分布。在这里表示网页变化的时间间隔符合指数分布，即：

$$\phi(t)=\begin{cases}\mu e^{-\mu t}, t>0\\ 0, t\leqslant 0\end{cases}$$

概率分布可以这样理解，在泊松过程中，下一个事件发生的时间间隔不超过 t 的概率为 $\int_0^t \phi(t)$。这里要注意概率和概率分布的区别。

对 $\varphi(t)$取自然对数，得到 $ln(\varphi(t))= -\mu t ln(\mu)$，因此 $ln(\varphi(t))$是一个关于 t 的线性函数，如上图 2-13 所示。直观上可以看到，网页变化的可能性随着变化周期（时间间隔）变长越来越小，文献[CHO, J. AND GARCIA-MOLINA, H. 2000c]较完整地论述

了关于变化间隔的计算方法及其他有趣的话题。

公式中的 t 表示网页发生变化的时间间隔，求 t 的期望寿命：

$$E(t)=\int_0^{\infty} t*\mu*e^{-\mu t}dt=\frac{1}{\mu}$$

如果把平均的时间间隔看做是网页寿命，则 μ 的倒数就是网页的期望寿命。

在泊松模型的理论基础上，结合人们的直观感知，目前网页重访策略大致可以归为以下两类。

（1）统一的重访策略：爬虫以同样的频率重访已经抓取的全部网页，以获得统一的更新机会，所有的网页不加区别地按照同样的频率被爬虫重访。

（2）个体的重访策略：不同网页的改变频率不同，爬虫根据其更新频率来决定重访该个体页面的频率。即对每一个页面都量身定做一个爬虫重访频率，并且网页的变化频率与重访频率的比率对任何个体网页来说都是相等的。即对于任意网页 P_i，其固有的更新频率 λ_i（次数/单位时间），爬虫对其重访的频率为 f_i，则 $\frac{\lambda_i}{f_i}$ 为一个常数。即对于网页 P_i 和 P_j，有 $\frac{\lambda_i}{f_i}=\frac{\lambda_j}{f_j}(i\neq j)$。那些更新频率高的网页，重访频率就高；更新频率低的网页，重访频率就低。

两种重访策略各有利弊，深入的研究表明对于那些更新频繁的网页采用更加频繁的重访策略反而有害[CHO, J. AND GARCIA-MOLINA, H. 2000b]。因此策略（2）还需要增加一些其他子策略，例如限定最大重抓频率等。策略（1）的效果在数学上可以证明总是不低于个体重访策略[CHO, J. AND GARCIA-MOLINA, H. 2000b]，并且具有计算量小的优势。但是策略（1）不具有优化的空间，难以满足质量不断改善的要求。

此外，不同类型的网站更新频率也不同。通过对不同类型的网页寿命的半衰期实验证明了这一点，网页更新的泊松分布如图 2-14 所示[CHO et al.2000c]。

通过该图，得到.com 类型的网页变化最剧烈，.gov 类型的网页变化最少，因此给予不同类型的网页不同的重访频率是科学的。

回顾网页更新策略既能保证抓取历史的网页，也能够抓取随时出现的新网页，

几个重要的结论如下。

（1）网页更新过程符合泊松过程。

（2）网页更新时间间隔符合指数分布。

（3）对于不同类型的网页采用不同的更新策略。

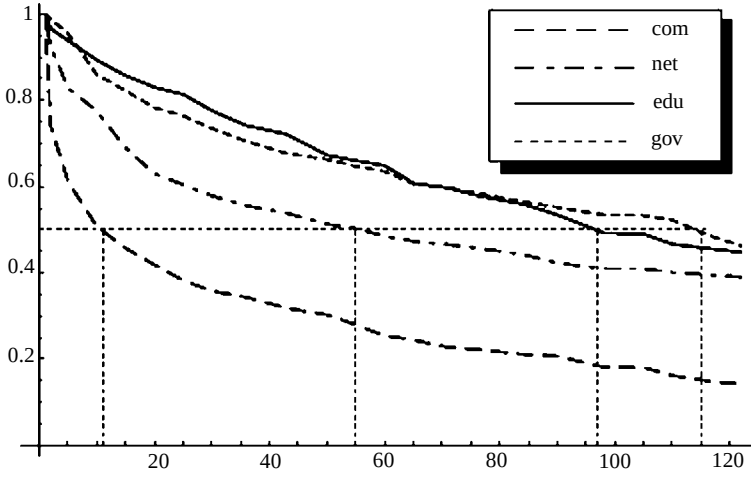


图 2-14　网页更新的泊松分布

粗放的工作做完了，然而还远远没有达到完美。爬虫的工作模式和方法存在一些问题，甚至可以说是社会问题 [BRIN, S. AND PAGE, L. 1998]。爬虫的工作方式某些方面上看是脆弱的，不可控的。因为在其中包含了数以千计的万维网上的 Web 主机，以及大量的 DNS 服务器，这些都是爬虫所不能控制的。只有协调好这些外部资源，才能达到理想的抓取效果。

有些网站不希望爬虫抓取敏感信息；有些网站希望爬虫避免在白天对其网页进行抓取，从而不影响白天正常的对外公众服务；DNS 服务提供商也不希望大量的域名解析工作量被搜索爬虫的域名请求所占用。

爬虫的工作可能带来的社会性的问题，要求爬虫一方面像绅士一样工作，了解网站管理员的需要。例如有些 Web 主机不希望爬虫抓取某些敏感目录，或者网页，这样网站管理员必须有办法告诉爬虫哪些是不希望被抓取的网页或目录。爬虫也需要礼貌地按照对方的要求执行，这就是著名的 Robots 协议。另一方面，爬虫尽可能自身维护一台 DNS 解析的服务器，而减少访问公开 DNS 域名解析服务器的机会。

最后，抓取的工作尽可能放在夜间，从而减少 Web 主机在白天的访问压力。

不同 Web 主机的带宽不同，能够接受的抓取强度也不同。爬虫需要尽可能平衡地抓取 Web 主机上的网页，将集中抓取平衡化；反之，如果集中抓取某一个站点，很可能会把某个 Web 主机抓瘫痪，而影响 Web 站点的正常公众服务。

遵守 Robots 协议是爬虫礼貌工作的第 1 步。

2.4.6 Robots 协议

Robots 协议是 Web 站点和搜索引擎爬虫交互的一种方式，即将一个 robots.txt 的文件放在网站的根目录上，例如 http://www.w3.org/robots.txt。

该文件的内容如下：

```
# robots.txt for http://www.w3.org/
# $Id: robots.txt,v 1.45 2006/06/05 01:11:19 ted Exp $
# For use by search.w3.org
User-agent: W3C-gsa
Disallow: /Out-Of-Date
User-agent: Mozilla/4.0（compatible; MSIE 6.0; Windows NT; MS Search
4.0 Robot）
Disallow: /
# W3C Link checker
User-agent: W3C-checklink
Disallow:
# exclude some access-controlled areas
User-agent: *
Disallow: /2004/ontaria/basic
Disallow: /Team
Disallow: /Project
Disallow: /Systems
Disallow: /Web
Disallow: /History
Disallow: /Out-Of-Date
Disallow: /2002/02/mid
Disallow: /mid/
Disallow: /People/all/
Disallow: /RDF/Validator/ARPServlet
Disallow: /2003/03/Translations/byLanguage
Disallow: /2003/03/Translations/byTechnology
```

```
    Disallow: /2005/11/Translations/Query
    Disallow: /2003/glossary/subglossary/
    #Disallow: /2001/07/pubrules-checker
    #shouldnt get transparent proxies but will ml links of things like
pubrules
    Disallow: /2000/06/webdata/xslt
    Disallow: /2000/09/webdata/xslt
    Disallow: /2005/08/online_xslt/xslt
    Disallow: /Bugs/
    Disallow: /Search/Mail/Public/
    Disallow: /2006/02/chartergen
```

在上述文件中，Robots 协议通过 User-agent 和 Disallow 告知搜索引擎非公开目录和非公开网页，说明如下。

（1）User-agent:*：表示对一切搜索引擎爬虫有效，如果特别针对某个爬虫，则可以写明。

（2）Disallow:/ 2004/ontaria/basic：表示禁止抓取这个目录。

通过遵守 Robots 协议，表示出爬虫尊重和执行 Web 站点的要求。因此爬虫需要有一个分析 Robots 协议的模块，并严格按照 Robots 协议的规定只抓取 Web 主机允许访问的目录和网页。

2.4.7　其他应该注意的礼貌性问题

除了遵守 Robots 协议以外，通常搜索爬虫需要自身维护一个 DNS 域名解析的服务模块，以便减少对公开域名解析服务器的频繁请求，本书不展开讨论这部分内容。

此外，应尽可能合理地规划抓取强度。在白天尽可能地减弱抓取强度，例如每 10 秒抓取一次；在夜间 Web 主机访问量低，搜索爬虫可以适当增加抓取强度，例如每秒抓取一次。由于时差的原因，东半球的白天恰好是西半球的夜晚，因此白天可以加强抓取美国及欧洲站点的强度，夜间增加对本国站点的抓取强度。

即便如此，爬虫总不可避免会给其他万维网上的 Web 主机带来麻烦。因此站点抓取监控程序也是必不可少的，该程序记录每个站点的抓取流量，避免在偶然情况下因抓取强度太大而导致的各种问题。

主机带宽在网页服务提供商一段是有限的，在搜索引擎爬虫这一段也是有限的，

而需要抓取的新网页和每天重访的网页规模都是巨大的，如何保证更重要的网页能够优选抓取，使得在时效性上和权威性上获得满意的效果，就需要对重要性网页进行优先抓取。

2.4.8 重要性网页优先抓取策略

此前讨论的网页爬虫算法，如果我们认为需要对重要网页进行优先抓取，可以在抓取过程中选择适当的时机进行重排（reordering），重排的代价和网页下载的代价相比是很低的，而重排可以确保重要网页被优先抓取，可以简单地写成如下形式[J Cho 1998]：

```
Crawling algorithm(seedlinks)
{
       for each link in seedlinks {
            enqueue(url_queue,link); //初始化抓取队列
       }
       While(not empty(url_queue)){
            url = dequeue(url_queue); //取当前抓取链接
            page = crawl_page(url);   //抓取当前链接，并存为 page 结构。
            enqueue(crawled_pages,(url,page));
                                      //抓取后的网页入 crawled_pages 队列
            url_list = extract_urls(page);
                                      //提取抓取网页内的全部链接
            for each u in url_list
                  if ((u not in url_queue) and ((u,-) not in craled_pages)
                  enqueue(url_queue,u);
                                      //将提取的链接，加入到抓取队列中
       reorder_queue(url_queue);      //对抓取队列进行重排
       }
}
```

这段代码在原有抓取过程中，每抓取一个链接进行一次插入排序式的重排，但事实上也可以积累一定数量，做一次遍历，多个插入排序式的重排，减少遍历代价。

这是一种思路，但随着规模的扩大，种类的增多，待排网页情况差距很大，可以采取多队列的方式，队列间分时间片的方式来处理，上述代码可以改写为[J Cho 1998]：

```
Crawling algorithm(seedlinks)
{
       for each link in seedlinks {
             enqueue(url_queue,link);
                                   //初始化抓取队列
       }
       While(not empty(url_queue)){
             url = multi_dequeue(hot_queue, url_queue);
             page = crawl_page(url);
                                   //抓取当前链接，并存为 page 结构。
             enqueue(crawled_pages,(url,page));
                                   //抓取后的网页入 crawled_pages 队列
             url_list = extract_urls(page);
                                   //提取抓取网页内的全部链接
             for each u in url_list
                   if ((u not in url_queue) and
                       (u not in hot_queue) and
                      ((u,-) not in craled_pages)
                   If(u is hot_url)
                                   //如果链接是重要网页的链接
                         enqueue(hot_queue,u);
                   else
                         enqueue(url_queue,u);
       reorder_queue(url_queue);
       reorder_queue(hot_queue);

       }
}
```

注意在 multi_dequeue 的实现中，可以对不同的队列给予不同的优先策略。例如每抓 10 个 hot_queue 的网页后，抓一个 url_queue 的网页。这样改写后，爬虫的抓取队列出现了两条队列，一方面通过重排操作给同类型下更重要的网页更多的带宽资源，另一方面通过多队列方式（例子中为了方便只举了两类），每个队列对应一种重要类型网页，通过队列的优先级来给重要类型的网页更多的带宽资源。

一个爬虫无论如何优化，单机的带宽资源总是有限的，而网页的数量是巨大的。如此巨大规模的万维网，每天新增的大量网页需要及时地被抓取到。对于爬虫来说，一方面对历史的网页需要重抓；另一方面要及时抓到新增的网页，在如此沉重的工作负荷下，必须提高抓取速度才能满足这种需要。

2.4.9 抓取提速策略（合作抓取策略）

提速基本可以采用下面几种方法。

（1）提高抓取单个网页的速度。

（2）尽可能减少不必要的抓取任务。

（3）增加同时工作的爬虫数量。

事实证明，受到万维网发展水平的限制，第（1）种方法基本不可行，单个网页的抓取速度受到下载带宽的限制，在现有技术条件下很难任意提高；第（2）种方法难度很大，由于需要和万维网的变化保持紧密同步，所以冗余的抓取总是不可避免的，减少不必要的抓取会导致网页重访不及时，这样就不能快速同步目标网页的变化；第（3）种方法通过增加爬虫数量提高总体抓取速度是可行的。

在多个爬虫抓取的情况下，如何将工作量分解成为主要的问题，即要解决一个网页交给哪一个爬虫抓取？如果分工不明，很可能多个爬虫抓取了相同的网页，从而引入额外的开销。通常采用以下两种方法来进行抓取任务的分解。

（1）通过 Web 主机的 IP 地址来分解，使某个爬虫仅抓取某个地址段的网页。

（2）通过网页的域名来分解，使某个爬虫仅抓取某个域名段的网页。

如何选择这两种方案呢？

万维网在网络基础设施中按照 IP 地址来确定主机位置，IP 地址为点分十进制数，难于记忆。由此采用了域名对 IP 地址进行一次映射，由于域名对人友好，于是出现了一些问题，即存在多个域名对应同样的 IP 的情况，对于中小网站来说，通常采用这种方法提供不同的 Web 服务。这主要出于经济的考虑，因为可以只配置一台服务器。而对于大型网站，如新浪和搜狐这些门户网站通常采用负载均衡的 IP 组技术，同样的域名对应于多个 IP 地址，一方面提高系统健壮性，一方面做到了负载均衡。

鉴于多域名对应相同 IP 和同域名对应多 IP 的情况，通常的做法是按照域名分解任务。即只要保证不重复抓取大型网站的网页，小型网站即便重复抓取也可以接受的策略分配任务。这种分配方法将不同的域名分配给不同的爬虫抓取，某一个爬

虫只抓取固定一个域名集合下的网页。例如 sina.com.cn 会固定交给 spider1 抓取，zoujingsousuoyinq.cn 会固定交给 spider2 抓取，zoujingsousuoyinqbook.cn 会固定交给 spider3 抓取等。

这两种方案的主要区别可以通过下面两个例子进一步理解。

假定 zoujingsousuoyinq.cn 和 zoujingsousuo yinqbook.cn 是两个域名不同、但 IP 相同的网站，假定为 10.10.67.208。有这样两个网页，即 http:// www.zoujingsousuoyinq.cn /index.html 和 http:// www.zoujingsousuoyinqbook.cn/index. html，在经过域名解析后，实际上它们同为 http://10.10.67. 208/index.html。采用域名分解的方案，spider2 和 spider3 就会重复抓到这个网页。由于这两个站点信息较小，所以可以容忍因抓重而带来的损失。

反之，我们如果通过 IP 分配抓取任务的方案，例如 sina.com.cn（71.5.7.138）被“指定”交给 spider1 抓取，sina.com.cn（71.5.6.136）被“指定”交给 spider2 抓取，zoujingsousuoyinq.cn（10.10.67. 208）被“指定”交给 spider3 抓取，zoujingsousuo yinqbook.cn（10.10.67.208）被“指定”交给 spider3 抓取。这种分配方案，对于不同域名指向相同 IP 的情况，例如 zoujingsousuoyinq.cn 和 zoujingsousuoyinqbook.cn 的抓取工作都由 spider3 完成，没有重复抓取问题。但是 sina.com.cn 对应多个 IP，被分配由 spider1 和 spider2 分别抓取，这样 spider1 和 spider2 抓取工作就相互重复。很显然，新浪属于大型网站，这种因为抓重带来的损失往往是巨大的。

通过比较，为了照顾大站按照域名分解的策略更加合理。在下载系统中，按照域名分解抓取任务（网页集合）的工作由一个称为“调度员”的模块来处理，通过域名分解将不同的网页调度给不同的爬虫进行抓取。因此，下载系统主要由爬虫和调度员构成。

形式化的调度分配方式如下：

首先假定有 n 个爬虫可以并行工作，并且定义一个可以提取 URL 域名的函数 domain，例如：

```
URL = http://finance.sina.com.cn/g/20070317 /17163416577.shtml
domain(URL)= finance.sina.com
```

说明如下：

（1）对于任意的 URL，利用 domain 函数提取 URL 的域名。

（2）用 MD5 签名函数签名域名，MD5(domain(URL))。

（3）将 MD5 签名值对 *n* 取模运算，int spider_no = MD5(domain(URL))%*n*。

（4）该 URL 分配给编号为 spider_no 的爬虫进行抓取。

由于模运算可以实现将一个全集分成多个等价类，所以等价类的并集等于全集，且一个等价类中的元素必然不属于另一个等价类中。

等价关系形式化表述如下。

令 $\boldsymbol{U}$ 为全集，通过某种等价关系可以将集合 $\boldsymbol{U}$ 分别映射到集合 $\boldsymbol{S}_1$，$\boldsymbol{S}_2$，…，$\boldsymbol{S}_n$ 上，并且满足如下两个条件：

（1）$\boldsymbol{S}_1 \cup \boldsymbol{S}_2 \cup ... \cup \boldsymbol{S}_n = \boldsymbol{U}$ 。

（2）$\forall a \in \boldsymbol{S}_i$，$\forall b \in \boldsymbol{S}_j$，$\boldsymbol{S}_i != \boldsymbol{S}_j$，则有 $a!=b$。

通常 *n* 取 2 的整数幂，例如 4，8，16，32……取模时可以利用按位与（&）的方法快速求得，int spider_no = MD5(domain(URL))&(*n*-1)。通常对 2 的整数幂 *n* 取模，都可以用&(*n*-1)的方法（其中 *n* 必须是 2 的整数幂才可以进行这样的计算）来快速计算。

这种策略的好处在于每个爬虫的任务量尽可能地均匀分配，同一个域名必然只由一个爬虫抓取，所有爬虫的工作量组合就是全部的抓取任务。在介绍了爬虫和调度员之后，已经能够完整地理解搜索引擎下载系统的体系结构，如图 2-15 所示。

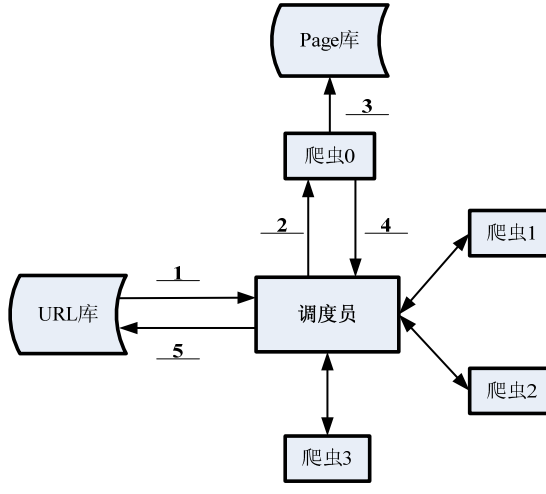


图 2-15　搜索引擎下载系统的体系结构

这里我们引入了 URL 库和 Page 库这两个概念，URL 库存放全部历史上曾经抓取过的 URL 和新增的 URL，Page 库存放爬虫实际抓取下来的原始网页。这样完整的合作抓取过程如下，注意下面的标号对应于上图中的数字。

（1）调度员通过更新规则向 URL 请求一个 URL 抓取任务。

（2）调度员计算出该 URL，然后分配给编号为 0 的爬虫抓取。

（3）爬虫 0 实际抓取的网页存放在 Page 库中。

（4）爬虫 0 在抓取的网页中提取其他链接后反馈给调度员。

（5）调度员判断网页类型，并设定初始更新时间等后存放在 URL 库中，继续转（1），周而复始。

在实际应用中，多采用多爬虫多调度员的体系结构，如图 2-16 所示。

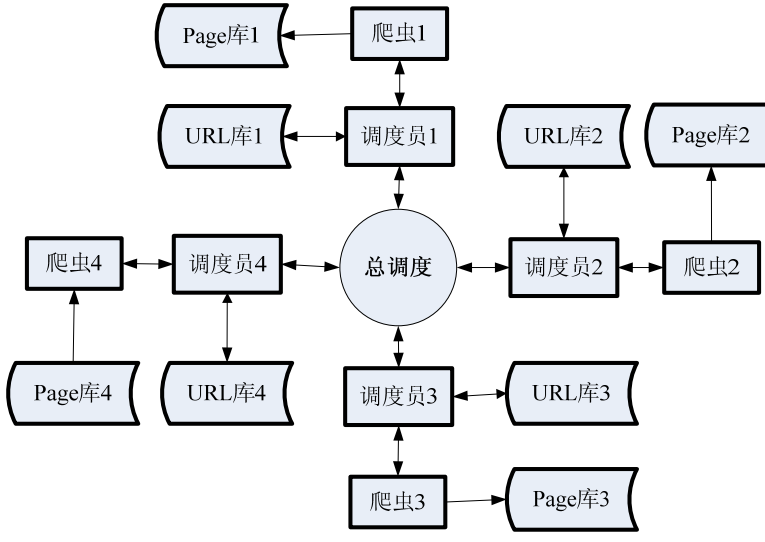


图 2-16　多爬虫多调度员的体系结构

值得注意的是，抓取的封闭性越强，对外的通信开销越小。如上图所示，假如爬虫 1 从 sina.com 这个种子站点开始抓取，由于总是抓取 sina.com 的网页，而这些网页总是应该归属爬虫 1 抓取，因此不需要和其他爬虫通信（不需要经过总调度）；反过来，如果抓取的封闭性差，表示可能抓到各种各样域名下的网页，并且可能需要交给其他爬虫抓取，总调度不得不把这些信息相互转发，这样就会增加额外的通

信代价。因此提高抓取的封闭性可以减少这种合作抓取带来的通信开销，前面提到过的宽度优先的遍历方法及深度策略能够有效的保证这种抓取的封闭性。

最后，简单了解一下大规模网页存储的有关知识。

2.5 网页库

很显然，爬虫抓取的网页必须及时保存在硬盘上，因此网页库的首要挑战来自于能够快速存储大规模网页；其次这些网页必须能够被其他模块快速地读取，因此读写问题是围绕网页库的主要难点。

（1）可伸缩性（Scalability）

网页库的存储必须具有可伸缩性，可以将大规模网页平滑地分布在一组计算机的硬盘中。

（2）双访问模式（Dual access modes）

网页库必须能够有效地支持两种不同的访问模式，即随机访问模式和顺序访问模式。使用随机访问模式，任意给出一个网页的标识即可读取相应的网页；使用顺序访问模式，顺序读取全部网页或者一部分网页。随机访问模式主要为搜索系统的查询系统中网页缓存部分提供随机的读取需要（在第 5 章中还会提到）；顺序访问模式主要用在索引系统中顺序读取网页按序索引的过程中。

（3）大规模更新（Large bulk updates）

万维网变化瞬息万变，网页库必须能够在网页删除后删除老版本的网页，如此处理可能会留下存储空洞。更新可以理解为删除后添加，而添加使用顺序添加到网页库的方法，这样不得不采用一些磁盘空间紧缩（space compaction）技术回收那些存储空洞。此外更新和访问还要保证互斥，避免同步出现的错误。

为了照顾这些特性，网页的存储方式大致分为以下 3 种类型。

（1）日志结构这（Log-structured）。

（2）基于哈希的结构（Hash-based）。

（3）哈希日志（Hashed-log）。

日志结构将磁盘看做一个连续存储的介质（可以想象为磁带机），这种存储方式只能顺序读及顺序写，因此存储介质对网页增加的操作非常有利。然而对于随机访问，则必须通过 B-+树的索引方式，因此每次访问需要执行两次读操作，分别读取索引和目标网页。此外还需要维护索引的代价，因此日志结构不能理想地支持随机访问的需要。

基于哈希的结构将磁盘看做一个哈希桶（Hash bucket）的集合，每个集合都足以存放在内存中，不需要构建额外索引，因此对于随机访问非常有利。然而对于顺序读写将十分不利，因为哈希函数的不确定性，对于一次网页增加，往往读入内存一个哈希桶，然后写入新增的部分，再写入磁盘。对于那些哈希桶中不变的数据，这样的一次读写操作无疑是浪费的，不断新增的网页被哈希映射到同一个桶的概率极低。还是因为哈希函数的不确定性，这种频繁读入哈希桶、更新并写入的操作被不断重复，因此对于网页新增是极其不利的。

哈希日志的结构是将哈希和日志的优势相结合，哈希作为索引找到文件块后再顺序写入，如图 2-17 所示。

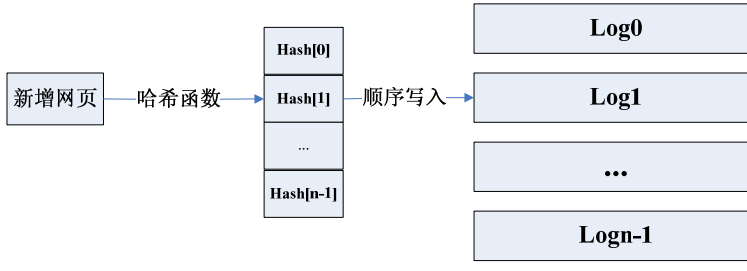


图 2-17　哈希日志的网页存储结构

对于新增的网页，通过 URL 计算出该网页的签名。进而通过模运算之后，将一个网页映射到哈希表的一个单元上，每个哈希表的单元对应于一个日志文件的位置。这个新增网页通过哈希函数的计算映射到 Hash[1]上，继而顺序写入 Log1 这个文件中。如果要随机读取某个 URL 的已存储的网页，或者仍然通过类似的哈希函数计算映射到具体的日志文件上。然后通过读取该日志文件上的 B-树索引，进而读取相应的网页文件。因此可以取得与日志文件等价，甚至稍好的随机访问效果（随机访问的目标文件大大减小）。最值得一提的是，哈希日志的方式可以支持批处理写入，从而大大改进了纯哈希结构。在每个日志文件中增加一个写入队列，只有积累一定数

量的文件后，才批量写入，如图 2-18 所示为增加排队机制的批量写入方法。

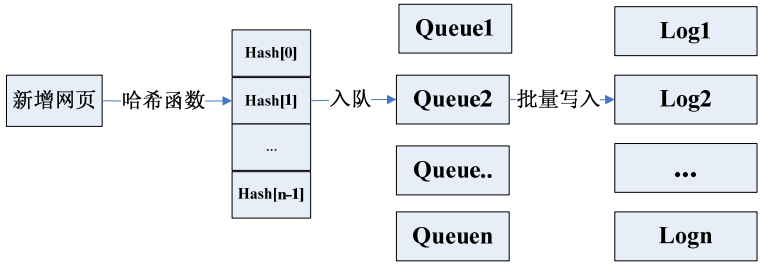


图 2-18　哈希日志批量写入记录

如上图所示，通过一个哈希表可以将那些新增网页插入的不确定性（与哈希结构相比）变为确定性插入，因此增加一个插入队列可以批处理方式插入到目标日志文件中。由于通过哈希函数的分解，所以在基于哈希结构的日志中每个日志的大小远小于日志结构中的日志，同时远大于哈希结构中的哈希桶。另外还必须保证每个日志也足够存放在内存中，因此确定哈希日志中哈希表的大小在很大程度上还需要考虑实际物理内存大小和需要存储的网页规模。

文献[Arvind Arasu et al. 2001]中给出了对于这 3 种网页存储方式的一个定性评价，如表 2-1 所示。

表 2-1　3 种网页存储方式的一个定性评价

	日 志 结 构	基于哈希的结构	哈 希 日 志
顺序访问	++	−	+
随机访问	+−	++	+−
增加网页	++	−	+

只要不是存在大量的随机访问机会，日志结构被认为是最佳的网页存储方式。对于可能存在大量的随机访问，以及大量新增网页的需要，哈希日志被认为是较为理想的网页存储方式。此外哈希日志能够有效地支持分布式的网页存储，在更大规模的环境中采用多机存储网页的情况下，它能够有效地将网页存储分布到每个存储结点上，增加网页存储的可靠性和稳定性。即便在某个存储结点上出现异常，对整体的搜索效果而言也不会产生较大的影响。

综上所述，网页库是搜索引擎中重要的一环，网页库具有不同的访问需求（顺序访问及随机访问），具体问题具体分析时需要结合这些需求选择相应的存储方式。在实践中，日志结构和哈希日志是较为常见的选择。

2.6　下载系统回顾及未来发展

通过系统地学习，至此终于揭开了搜索引擎下载系统的神秘面纱。实现下载系统的主要需求，总结下来最主要是以下 3 点。

（1）**抓得全**：通过网页更新策略（更新目录型网页）。

（2）**抓得快**：通过合作抓取策略。

（3）**代价低**：通过宽度优先的遍历策略、最大深度策略及合理的网页更新策略。

当然这与一个实际的商用大型搜索引擎下载系统还有很大距离，例如大规模网页存储、DNS 服务模块、反垃圾、反病毒、多爬虫的协调工作，以及爬虫监控等。读者如果亲自实现一个爬虫，还会遇到很多困难。目前比较著名的关于爬虫的开源代码很多，例如 Web sphinx 和 Heritrix 等。读者可以在学习本章内容后，实际参与到这些开源代码的学习和使用中，以进一步加深对搜索引擎的下载系统的认识。

[M.P.S.Bhatia 2008]提到目前由于商业原因，最新的爬虫技术往往秘而不宣，其中提到了 Internet Archive 公司关于爬虫的一些信息，摘录如下：Internet Archive 使用多机合作抓取网页的方式，每个爬虫进程只分配 64 个站点，每个站点只分配给一个爬虫，每个单线程的爬虫进程读取一个种子 URL 的列表，并且为每个站点建立一个 Queue，接下来使用异步 IO 的方式从这些队列中并行的抓取网页。当一个网页下载完成后，爬虫提取该网页包含的链接，如果该链接是 64 个管辖站点时，则该链接将会放在 64 个 queue 中的某一个中，否则这些链接会存放在磁盘中，同时系统定期将这些磁盘中的跨站链接(cross-site URL)归并到管辖这些爬虫的队列中，并同时做相应的去重工作。

以上可能是目前商用搜索引擎能够公开的最为准确的描述，其中包含了任务分配的方法、下载的方法和跨站链接的处理方法。其中异步 IO 的方法，[刘奕群 et al，2010]5.3.2 小节对异步 IO 有专门小节进行论述，并且在本书优化一章中还会进行详细讨论。

虽然前人做出了巨大的努力，取得了大量的成果，然而爬虫的发展还远远没有停止。未来还有很多工作等待我们去探索，那么还有哪些工作可以做呢？

（1）动态网页支持

Web 上动态网页是静态网页的 400～500 倍，光明星球[brightP]公司宣称，存在的网页总数至少为 5 500 亿个，这个数量是相当惊人的。目前几乎所有的搜索引擎都不能完全解决抓取动态网页的难题，因为这些动态网页通常都受到了账号和密码的保护，这也称为“深度挖掘问题”。

（2）定向抓取

定向抓取通常也称为“聚焦爬虫”，目的是使爬虫的工作方式不再是漫无目的，而是在某种意图下抓取有价值且特定的网页。聚焦爬虫的评价一般采用收获率（harvest ratio）作为评价，即采集的符合需求的网页与采集的全部网页之间的比率。

参考文献

[Arvind Arasu et al. 2001] Arasu, A. and Cho, J. and Garcia-Molina, H. and Paepcke, A. and Raghavan, S. Searching the Web, ACM Transactions on Internet Technology,42 pages.

[Bloom, B.H. 1970] Space/time trade-offs in hash coding with allowable errors Communications of the ACM, 1970.

[BrightP]TheDeepWeb: Surfacing Hidden Value. http://www.brightplanet.com/deepcontent/tutorials/DeepWeb/index.asp.

[Broder, et al.,2000] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and a. J. Wiener, "Graph structure in the web: experiments and models," presented at Proceedings of the 9th World-Wide Web Conference, Amsterdam, 2000.

[BRIN, S. AND PAGE, L. 1998] The anatomy of a large-scale hypertextual Web search engine.Comput. Netw.

[CHO, J. AND GARCIA-MOLINA, H. 2000a] Estimating frequency of change. ACM Transactions on Internet Technology, Vol. 3, No. 3, August 2003.

[CHO, J. AND GARCIA-MOLINA, H. 2000b]. Synchronizing a database to improve freshness. In Proceedings of the ACM SIGMOD Conference on Management of

Data. ACM Press.

[CHO, J. AND GARCIA-MOLINA, H. 2000c] The evolution of the web and implications for an incremental crawler. In Proceedings of the 26th International Conference on Very Large Data Bases.

[HIRAI et al 2000]HIRAI, J., RAGHAVAN, S., GARCIA-MOLINA, H., AND PAEPCKE, A. 2000. Webbase: A repository of web pages. In Proceedings of the Ninth International Conference on The World Wide Web. 277–293.

[Jeff Heaton,2002] Programming Spiders, Bots, and Aggregators in Java Jeff Heaton Publisher: Sybex，February 2002，ISBN: 0782140408, 512 pages.

[J Cho 1998]Efficient Crawling Through URL Ordering.J Cho, H Garcia-Molina, L Page - Computer Networks and ISDN Systems, 1998 – Elsevier.

[M.P.S.Bhatia 2008] Discussion on Web Crawlers of Search Engine. In Proceedings of 2nd National Conference on Challenges & Opportunities in Information Technology (COIT-2008) RIMT-IET, Mandi Gobindgarh. March 29, 2008.

[Reka Albert et al. 1999]　Reka Albert, Hawoong Jeong and Albert-Laszlo Barabasi. Diameter of the World-Wide Web. Nature 401, 130-131, Sep. 1999.

[刘奕群 et al，2010]，马少平，洪涛，刘子正，“搜索引擎技术基础”.

[Li02]　　李晓明，“对中国曾有过静态网页数的一种估计”，PKU_CS_NET_TR2002006, 2002 年 4 月.

[闫宏飞,2002] 闫宏飞，“可扩展 Web 信息搜集系统的设计、实现与应用初探”北京大学，博士论文，2002, pp. 116.

[闫宏飞 and 李晓明,2002]　闫宏飞，李晓明，“关于中国 Web 的大小、形状和结构”计算机研究与发展，vol. 39，No.8，2002.

第 3 章　搜索引擎的分析系统

3.1　知识准备

3.2　信息抽取及网页信息结构化

3.3　网页查重

3.4　中文分词

3.5　PageRank

3.6　分析系统结构图

3.1　知识准备

搜索引擎的 4 大系统中的第 2 个系统是分析系统，分析系统主要完成的工作包括信息抽取、网页消重、中文分词和 PageRank 计算等。接下来的各节将按照这个顺序层层深入地介绍分析系统的工作，在此之前了解一些基本概念。

3.1.1　HTML 语言

HTML 语言（Hyper Text Markup Language，超文本标记语言）是一种专门的编程语言，用于创建存储在 WWW 服务器上的文件，并能由 Netscape 及 Microsoft Explorer 等浏览器浏览。由于 HTML 语言简单易用、不需要编译的特性，因此被广泛地使用。

3.1.2　锚文本（anchor text）

锚文本网页中关于链接的一段描述，通常以文本和图片的方式出现。可以指向文中的某个位置，也可以指向其他网页。例如，HTML 脚本走进搜索引擎中的“走进搜索引擎”就是一个“锚文本”，它用来描述其指向的链接。

3.1.3　半结构化数据（semi–structured data）

和普通纯文本相比，万维网上的网页数据具有一定的结构性，但是这种 HTML 标签带来的结构性不能满足网页结构化的需要。例如<TITLE>标签标识网页主题，而<TD>标签有些表示文章主题，有些表示文章段落，或者其他广告信息等。因此人们称网页的原始数据为“半结构化数据”，这是 HTML 语言的基本特点。

3.2　信息抽取及网页信息结构化

对于分析系统来说，基础和首要的工作是分门别类地从半结构化网页

（semistructured page）中抽取（extract）出有价值的能够代表网页的属性，例如锚文本、标题和正文等，并将这些属性结合起来组成一个网页对象，这种处理称为“网页结构化”。

3.2.1 网页结构化的目标

网页结构化的目标是针对搜索的需要，将半结构化的 HTML 网页中的数据按照如下几个基本属性依次抽取，最后打包（wrap）出一个网页对象。

（1）锚文本（anchor text）。

（2）标题（title）。

（3）正文标题（content title）。

（4）正文（content）。

（5）正向链接（link）。

网页的这 5 个基本属性对于检索来说至关重要，下面一一介绍。

（1）锚文本：除了网页标题可以描述网页以外，还会有一些锚文本来描述它。例如清华大学主页可能被另外一些网页中存在锚（anchor）所指向，其锚文本就是该网页的最佳描述。特别是对于某些没有标题的网页，锚文本是有益的补充。

（2）标题：这里的标题特指 HTML 标识语言中<title></title>中间的文字部分，这部分文字表达了网页的基本含义。和锚文本相同的是，都是用来描述网页的内容的属性；和锚文本不同的是，这个标题是由该网页制作者本人编写的。

（3）正文标题：在 HTML 网页中，网页的标题由<title>标签标识。实际的情况是由于网页编写者的疏忽，或者其他原因，<title>标签中的文字没有任何网页描述能力，并不是合格的标题，为此需要抽取正文中的适当文字作为正文标题。

（4）正文：锚文本、标题和正文标题都是网页的简短描述，而正文是一个网页的主体内容，它完整地表述了网页的主体内容，一般出现在<DIV>、<TABLE>和<P>等 HTML 标签中。

（5）正向链接：正向链接是网页制作者编写的引导用户继续在网上冲浪的链接，

这些链接的文字也是其他网页的锚文本。

一个网页对象（Page Object）简单说来至少包含 5 个属性，其含义如表 3-1 所示。

表 3-1　网页对象的 5 个属性及其含义

属　　性	含　　义
title	页面标题
content title	正文标题
content	正文
link	正向链接集合
anchor	指向自身的锚文本

在不失完整性的前提下，不妨对问题进行简化，通过结构化网页中标题和正文两种网页对象的属性来理解网页结构化的基本原理。

假定下面这个简单 HTML 网页的文件名为“book.html”：

```
<HTML>
<HEAD><TITLE>走进搜索引擎</TITLE> </HEAD>
<BODY>
<TABLE>
<TR><TD>走进搜索引擎：第一章</TD> </TR>
<TR><TD>走进搜索引擎：第二章</TD> </TR>
<TR><TD>走进搜索引擎：第三章</TD> </TR>
</TABLE>
</BODY>
</HTML>
```

将以上内容写入文件 index.html，用 IE 浏览器打开，显示如图 3-1 所示。

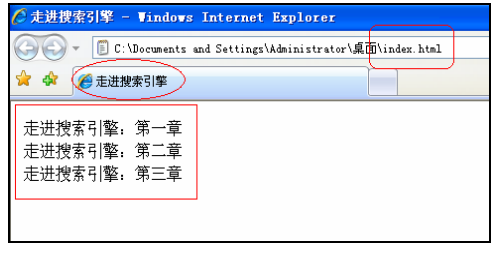


图 3-1　包含标题和正文的简单网页示例

圆角矩形框表示该网页的文件名；椭圆框表示网页的标题（TITLE）；矩形框表示该网页正文，其中包含几段文字。

网页结构化的过程可以直观地理解为如图 3-2 所示的过程。

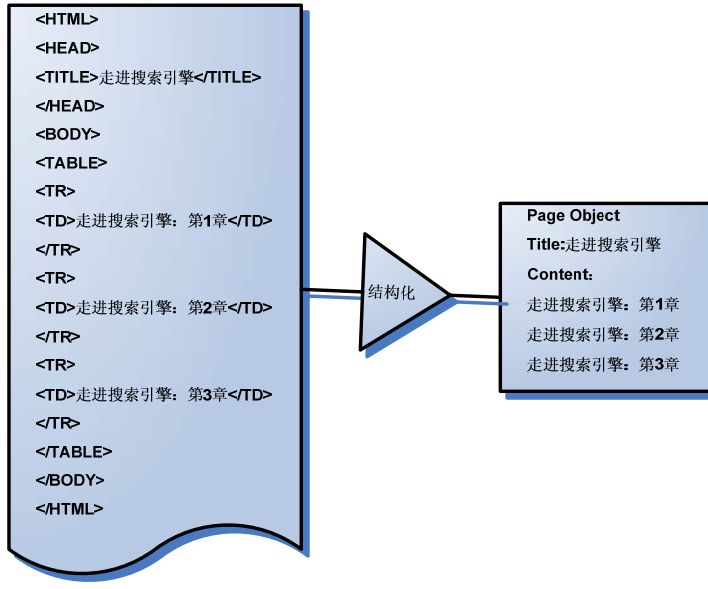


图 3-2　网页结构化的过程

在理解结构化网页的目标后，加上对 HTML 语法的特点，以及搜索引擎的实际需要，为这种分析原始网页的过程制定了如下“两步走”的方法。

（1）建立 HTML 标签树（tag-tree）。

（2）通过投票方法识别正文的文本块，并按照深度优先遍历的规则组织为正文。

3.2.2　建立 HTML 标签树

万维网上大多数的静态网页都以 HTML 网页形式存在，HTML 是一种标识语言（Markup Language），它把其描述的全部内容都按照 HTML 语法存放在标签之中。为了更清楚地描述网页内容的组织结构，将网页中的标签按照出现顺序依次整理出来并用适当的结构记录。由于标签之间的嵌套关系，因此整理结果自然是一个树状结构，我们把整理一个网页中的标签得到的树状结构称为该网页的“标签树”。

很明显，浏览该网页的用户看到的是相当友好的信息。而实际源文件中的那些 HTML 标记，如<HTML>和<TABLE>（可以理解为用来帮助 IE 浏览器理解网页）等都不会实际地展示给用户。因此分析系统需要学习 IE 浏览器理解网页的方式来理

解网页，在理解过程中需要建立一个 HTML 标签树的树型结构，如图 3-3 所示。

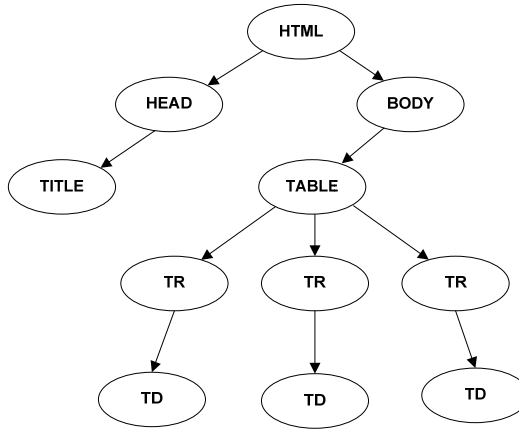


图 3-3　HTML 标签树

HTML 语法中每种标签都是成对出现的，这有助于标记一个标签的始末，因此需要一种能够记忆历史的数据结构来协助标签树创建和分析过程。下面通过一个例子来理解这个标签树建立的过程。

首先，该过程需要一个标签分析栈的数据结构。栈结构是一种先进后出的线性表结构，栈结构的这种特性为分析工作提供了可能。通过简单的图形可以把实际的标签抽取工作容易地表达出来，如图 3-4 所示。

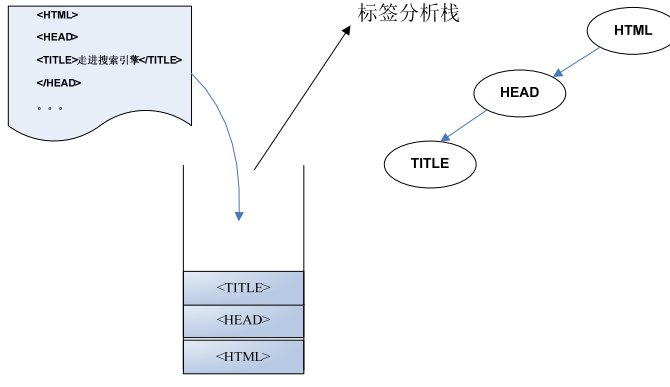


图 3-4　标签分析栈

文本读取程序将网页源文件中的<HTML>、<HEAD>和<TITLE>依次投入分析栈中，在右边依次建立一个标签树，栈底元素<HTML>为树根。当读取到“走进搜

索引擎”时发现该文本不是标签，于是将其保留。由于栈顶标签为 TITLE，因此该字符串为正文标题，如图 3-5 所示。

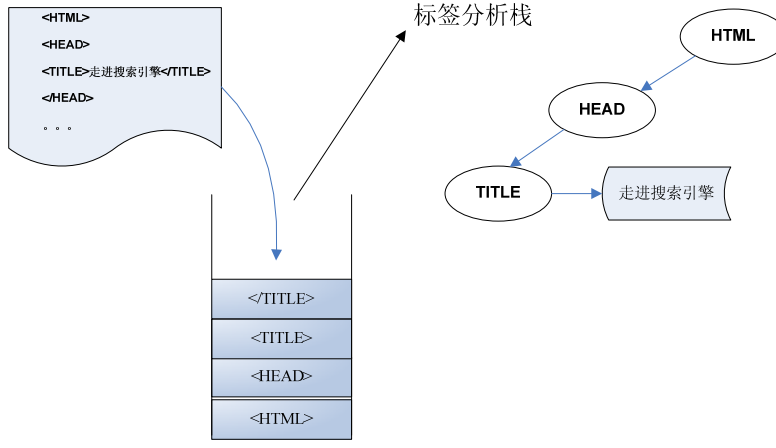


图 3-5　抽取 TITLE 标签所含文字

文本读取程序继续读出</TITLE>这个标签，并将其放入标签分析栈中。此时，标签栈顶的标签为<TITLE>，因此这两个成对标签同时退栈，如图 3-6 所示。同时也确认了“走进搜索引擎”这一字符串为正文标题，在标签树上的 TITLE 标签存放一个指向该字符串的指针，这样 TITLE 标签的文字被正确地抽取出来。

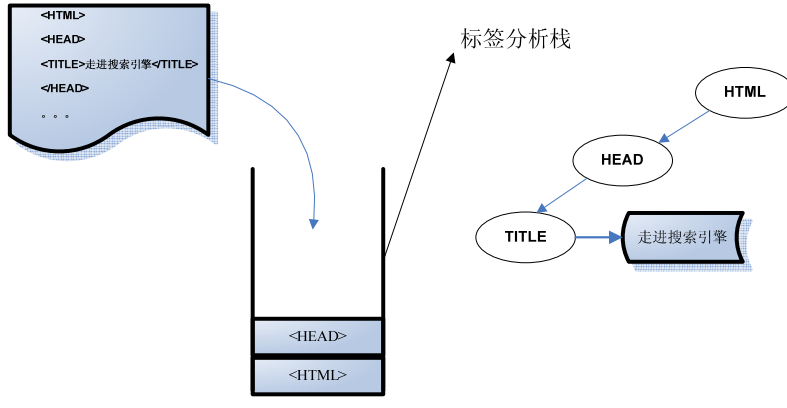


图 3-6　退栈过程

接下来继续读入</HEAD>标签后，两对标签同时退栈。下面的过程以此类推，直到<HTML></HTML>标签对退栈为止，整个分析过程结束后形成如图 3-7 所示的形态。

进一步了解 HTML 标签的含义及其分析容错的有关细节需要阅读相关文献，以及深入理解 HTML 语法并积累大量的实际经验，为了便于理解，在此略去了这些过程。

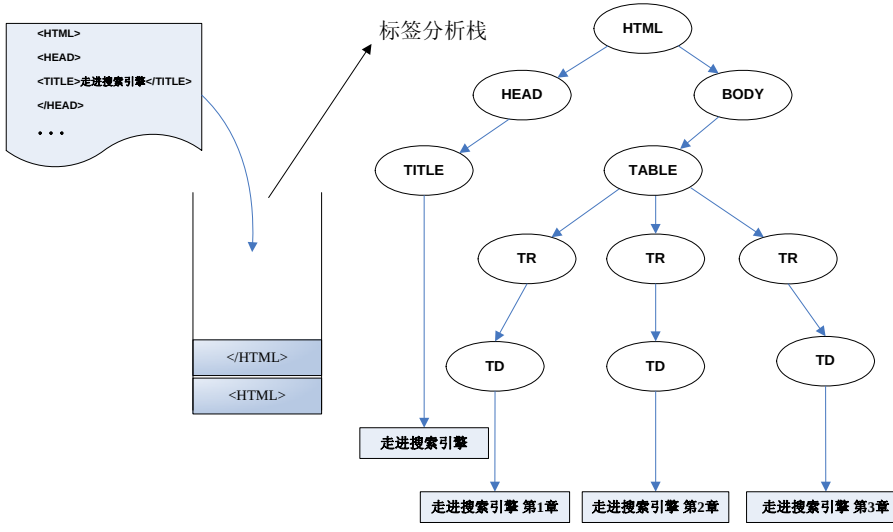


图 3-7　分析过程结束后形成的形态

通过建立标签树，并且识别标签所描述的文字，网页结构化的进程走出了重要的一步，如图 3-8 所示。

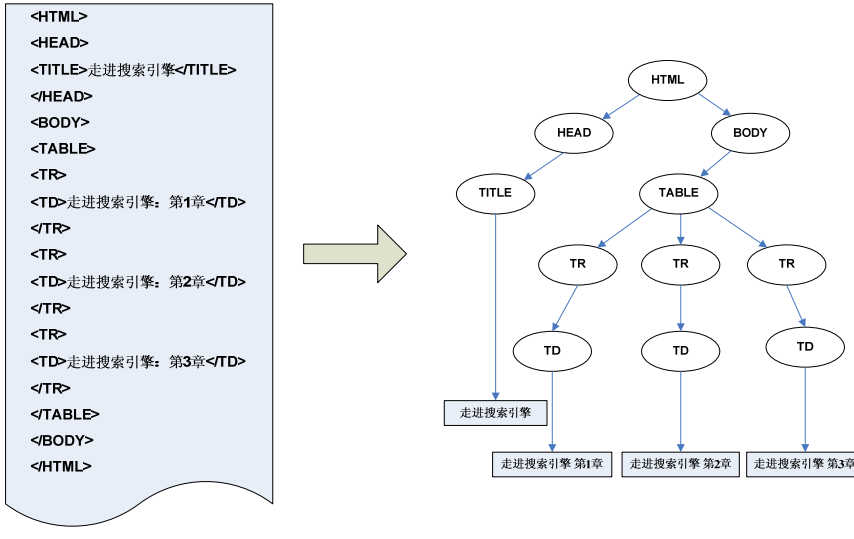


图 3-8　从 HTML 网页到结构化网页

通过上面这个过程已经能够提取出标题，很明显 TITLE 标签下的文字即为标题。进展令人振奋，然而离目标还有很大距离。在实际的网页中，正文、广告及其他各种信息都可能出现在<table>和<td>等标签中，哪一个才是真正的正文？由于正文可能分片，例如上面这个例子中的正文分别位于 3 个<TD>标签上，如何将其组合起来？

3.2.3 通过投票方法得到正文

由于网页半结构化的特性，一方面，TITLE 很容易地通过第 1 步标签分析得到；另一方面，得到完整的正文却非常复杂。首先，网页中没有明显的标签标识出正文；其次正文可能分散在多个 HTML 标签中，问题在于如何组合出完整的正文。首先来解决第 1 个难点，即识别正文。

正文具有分块保存的特性，因此我们引入文本块的概念，对于那些诸如<P></P>等标签间的文本认为是一个文本块。例如，“<TD>走进搜索引擎：第 1 章</TD>”称为一个文本块。

一般来说，网页会出现 3 种类型的文本块。

（1）主题型文本块（topic）。

（2）目录型文本块（hub）。

（3）图片型文本块（pic）。

主题型文本块是大段文字的文本块，例如：“<TD>走进搜索引擎：第 1 章</TD>”。

目录型文本块是描述链接的文本块，例如：“走进搜索引擎：第 1 章”。

图片型文本块是描述图片的文本块，例如：“<img src="">走进搜索引擎：第 1 章</img>”。

目录型文本块和图片型文本块相对容易被区分；而主题型文本块中可能包含广告等其他内容，必须与正文相区别。

判断哪个文本块是正文采用称为“投票算法”的计算方法，这种方法在搜索引擎中特别常用。

在日常生活中几乎所有人都会有投票或者选举的经历，例如选举干部和通过决议需要投票，以及运动员的一套动作需要裁判员打分等。其基本原理在于认为大多

数人的意见往往是正确的，大多数人的统一主观意见就会变得较为客观。例如在体操运动员结束一套动作之后，多个裁判根据自己主观意见给出独立的评判。然后去掉一个最高分和一个最低分，这时的分数尽管是主观产生的，但是这种评判的方法和结果被认为是相对客观和可信的。

正文抽取的投票算法的过程如何呢？首先我们会定义一系列规则，然后通过这些规则为每一个文本块打分。得分最高的被认为是正文的可能性足够大，并且可以接受。

我们来举这样一个例子，假定一个规则集合中包含以下 3 条规则。

（1）如果文本块文本的长度少于 10 个字，得分为 0；介于 10~50 个字得分为 5 分；介于 50~250 个字，得分为 8 分；超过 250 个字，得分为 10 分。

（2）如果文本块文本位置在右侧，得分为 0 分；在顶部，得分为 3 分；在左侧，得分为 5 分；在中间，得分为 10 分。

实际的规则往往更加繁多和复杂，为了介绍该方法，我们定义文本长度和文本位置规则。投票算法的过程是依据不同的规则从不同的角度依次打分，文本块得分高的是正文的一部分。投票算法的一般模型如图 3-9 所示。

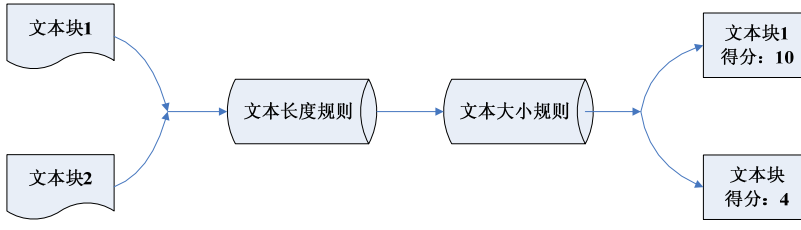


图 3-9　投票算法的一般模型

如图所示，每个规则器打分后，累积总分为文本块 1 得 10 分，远大于文本块 2 得到的 4 分，因此可以认为文本块 1 是正文内容的可信度高。

在实际的程序中，规则器的打分是不断调整的，其个数也可以支持动态添加。如果规则打分计算比较复杂耗时，则可以采用并行打分的方法。这样所需要的时间将由最耗费时间的规则计算代价决定；否则通过串行打分，时间耗费将会是所有规则计算代价的总耗时。为了简化，图 3-9 中采用串行打分的方法。

除此之外，规则的定义还需要通过足够多的网页进行反馈，之后才能得到一个公正客观的打分。这类似裁判员在打分前的业务学习，只有在比赛前透彻地理解评分标准，并且很好地观察运动员完成动作的情况。通过不断地训练才能区别动作的难易程

度，达到比赛裁判的水平。此外，有些动作在若干年后由于掌握的运动员很多，因此评分标准就会发生变化。新出现的高难动作将会得到评分上的青睐，规则器也需要类似这样的业务学习过程。如果经常发现某些网页的正文段抽取错误，则要找出是哪一个规则器打分不合理才会导致这个结果。反复这样的过程，最后的打分将会趋于合理。

最后还需要注意这样的问题，如果规则 A 打分为 0~1 000 分，规则 B 为 0~10 分。那么很显然，规则 B 对最后打分结果的影响力几乎可以不考虑，因此如何平衡规则对最后结果的影响力也需要充分考虑。这是因为规则的重要性不同可以适当拉开距离，但有时需要在可以接受的范围内保持这种距离。

即便做出了如此多的工作，投票的结果也并不总是正确的。真理也常常会站在少数人的一边，群众的眼睛也不可能总是雪亮的，做到百分之百的判断准确即便对人来说也都是异常困难的。因此采用这种方法永远不可能完全正确，只能是达到某种概率下的正确，而作为投票算法追求的目标是将这种得到正确结果的概率不断地提高。

剩下的工作就是如何将一个个文本块组织成一个正文？深度优先遍历标签树并依次记录主题类型的文本块，即可得到该网页的正文。为了说明深度优先遍历的方法，我们使用一个略微复杂的例子，如图 3-10 所示。

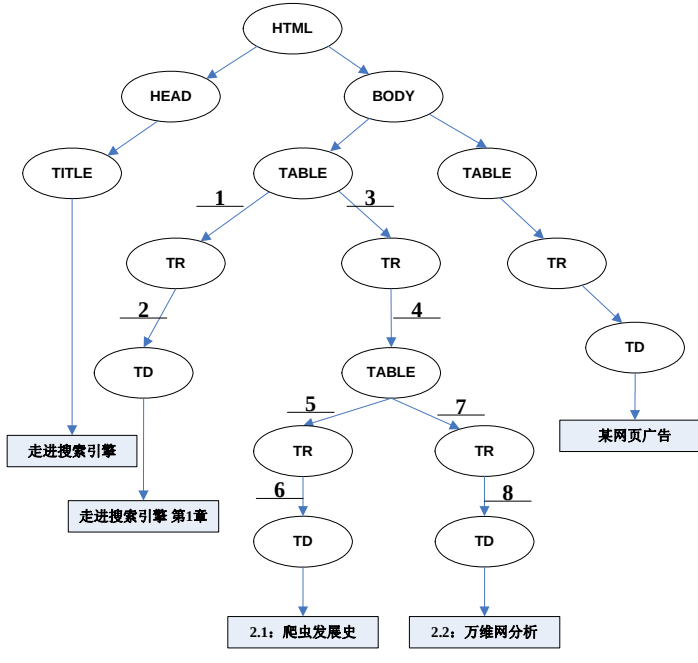


图 3-10　提取正文的方法

在图 3-10 所示例子中的 BODY 标签下，第 1 个 TABLE 为正文，因为其下的文本块通过投票打分均为文本块。而第 2 个 TABLE 标签下通过投票打分，其得分较低，因此可能为广告类信息。这样组织正文只需要从第 1 个 TABLE 开始深度优先遍历，遍历顺序为图中带下划线的数字所示，因此依次提取的文本块并按照顺序组织成如下的正文：“走进搜索引擎 第 1 章 第 2.1 节：爬虫发展史 第 2.2 节：万维网分析”。正是由于 HTML 标签的这种相互嵌套的特性，所以决定了深度优先遍历的顺序恰好能够组织成一个完整的正文。

对于其他的网页属性抽取，例如正文标题等也大多采用相同或类似的方法。

锚文本的提取有一些复杂，由于锚文本对网页的描述存在于其他网页中，例如南京大学的主页，被其他网页所链接，这些网页中包含了“南京大学主页”或“南京大学”等这样的锚文本，因此南京大学主页这个网页对象的锚文本就是“南京大学”、“南京大学主页”。锚文本提取，筛选等工作需要在获得足够多的网页后才能进行，采用分块计算的方法，比较复杂，这里不再展开叙述。

3.2.4　网页结构化过程回顾

网页结构化的过程首先通过标签树进行分析得到文本对应的标签，然后通过投票算法确定正文及配图等仅从 HTML 标签无法判断的网页数据。这样就圆满完成了结构化的任务，达到了理想的结构化要求，如图 3-11 所示为两步走过程。

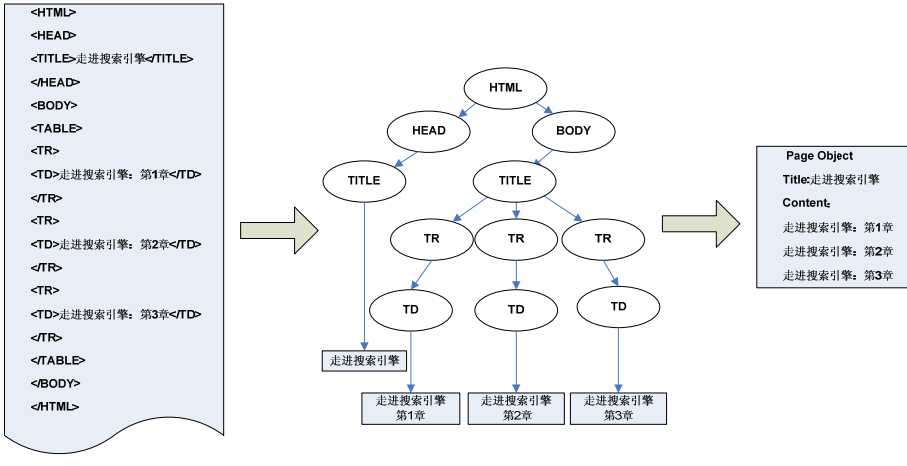


图 3-11　两步走过程

直观地感觉到网页结构化的过程使得网页有价值的信息被保留，例如标题和正文；而网页中无用的信息被丢弃，例如 HTML 标签。因此网页结构化的过程同时也节约了大量的存储，存储一个网页不再是存储原始的网页，而是只需要存储结构化的网页。即便是这样，存储和处理大规模网页依然是一项艰巨的任务，解决这个问题首先想到的是应该去掉那些相同或者类似的网页。这样不仅可以大大节约存储，还可以有效地提高检索效果，下一节将介绍有关网页查重的技巧。

3.3 网页查重

在日常上网时一般很少会留意到相同相似网页，一些偶然的机会也能看到一些相对重要的网页会被各大网站转载，或者在浏览 BBS 时会看到各种各样的转帖。对于网民来说，这种重复是有利的，便于大多数人看到重要的信息。然而对于搜索引擎来说，重复网页的存在意味着这些网页至少被多处理一次。更糟糕的是，在接下来的索引制作中可能会在索引库中索引两份相同的网页。当有用户查询时，在有限的查询结果页中（一个查询结果页显示 20 条网页链接）就会出现重复的网页链接。因此无论从系统效率，还是从检索质量来说，重复网页都是有害的。

3.3.1 网页查重技术发展历史

网页查重技术起源于复制检测技术，复制检测即判断一个文件的内容是否抄袭、剽窃或者复制于另外一个或者多个文件。

1993 年，Arizona 大学的 Manber 推出一个 sif 工具，用于在大规模文件系统中寻找内容相似的文件。1995 年，Stanford 大学的 Brin 和 Garcia-Molina 等人在“数字图书馆”工程中首次提出了文本复制检测机制 COPS（Copy Protection System）系统与相应算法[Sergey Brin et al 1995]。之后这种检测重复技术被应用到搜索引擎中，基本的核心技术即比较相似。

和简单的文档不同，对于网页来说，这种相似包含了特别丰富的含义。网页具有内容和格式，因此在内容和格式上的相同相似构成了如下 4 种网页相似的类型（如果把相同看做相似的一种特殊情况）。

（1）两个网页的内容和格式上完全相同（full-layout duplicates）。

（2）两个网页的内容完全相同，但格式不同（full-content duplicates）。

（3）两个网页有部分重要的内容相同并且格式相同（partial-layout duplicates）。

（4）两个网页有部分重要的内容相同，但格式不同（partial-content duplicates）。

如图 3-12 所示，搜索引擎在相似问题上按照内容和格式的组合共计 4 种不同的形式。对大规模网页来说，网页查重只需要找到内容相同或相似的网页，而忽视格式上的异同，因此在网页查重的过程中，图 3-12 所示的 4 种相似情况都属于网页查重的目标。

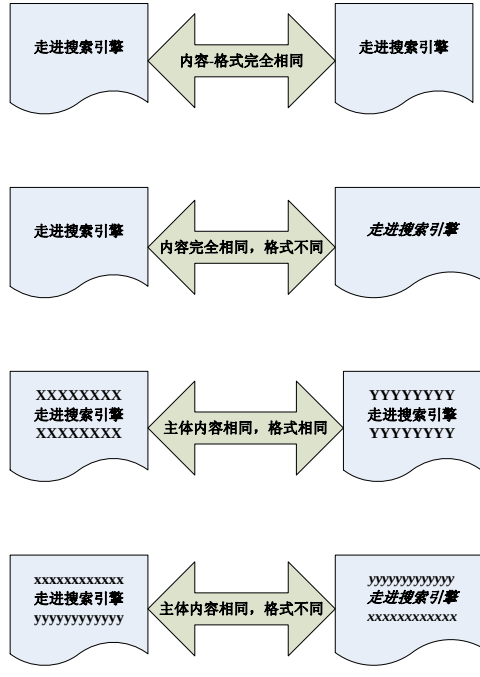


图 3-12　网页相似性的 4 种形式

采用网页签名的方法很容易检测网页完全一致。如果两个网页签名值相同，则为相同网页，这种方法在第 2 章中关于网页 URL 不重复抓取的策略中已介绍过。因此本节主要关注的是检测网页相似的情况。

检测大规模的相似网页需要考虑准确性和效率，文献[Abdur chowdhury et al 2002]，[Broder 1997]分别提出了两种经典的处理大规模网页相似检测的方法，下一节我们将看到这一有趣的方法。

3.3.2 网页查重实现方法

由于不考虑格式的异同，网页查重首先需提取结构化网页的正文和标题，将复杂的网页转化为具有标题和正文的文档，因此下面称网页查重为“文档查重”。

日常生活中判断两个相似事物一般采用比对特征的方法，特征是那些有代表性、不易变化、能够反应主体本质的信息。文档查重的第 1 步就是特征抽取，下面介绍两种特征抽取的方法。

第 1 种特征抽取方法着眼于尽可能抽取一个特征，这样比较两个文档是否相似只要比较一次即可。文献[Abdur chowdhury et al 2002]首先提出了称为“I-Match”的算法，这个算法基于一个假设，即一篇文档中特别高频和特别低频的词汇往往不能反映这篇文档的本质。因此通过将文档中去掉高频和低频词汇后的词汇通过排序得到一个字符串，使用签名算法获得该字符串的签名。如果有其他文档和这个签名值相同，则判定为相似。

I-Match 算法的示例如图 3-13 所示。

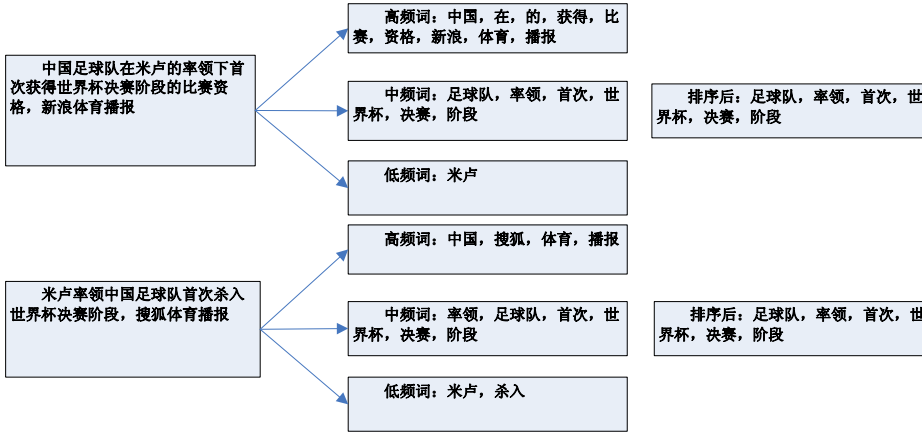


图 3-13　I-Match 算法抽取特征

如图 3-13 所示的例子中，通过去掉高频和低频词汇（词频的统计工作由索引系统提供，在第 4 章中会提到词频的统计方法和技巧），两个文档趋于一致，通过排序则得到完全相同的词汇列表。通过这种计算，两篇文档的相似性就转化为一个词汇列表是否相同的问题。

第 2 种特征抽取方法是抽取多个特征词汇，通过比较两个特征集合的相似度实现文档查重。抽取多个特征的方法很多，这里介绍其中的一个经典算法，称为“Shingle 算法”。Shingle 在英文中表示相互覆盖的瓦片。接下来通过上面的例子来说明 Shingle 算法为什么命名为“Shingle”。例如使用每 5 个汉字为一个 Shingle 的方法，如图 3-14 所示。

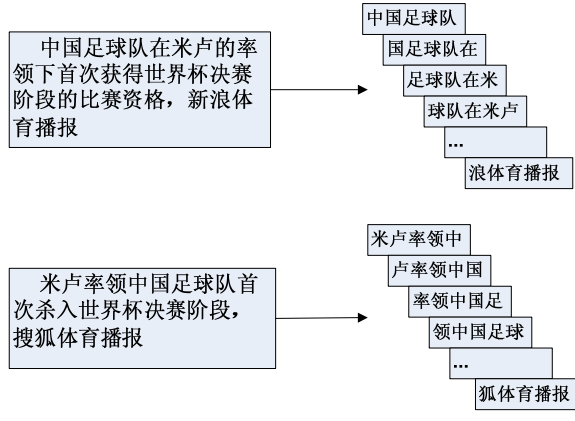


图 3-14　Shingle 算法抽取特征

如图所示，每 5 个汉字组成的 Shingle 恰如其名，好似相互覆盖的瓦片一样。由于把一个文档转化为一组字符串集合（每个元素为一个 Shingle），因此判断两个文档的相似性就转化为字符串集合的相似性。

最后介绍一个关于 Shingle 算法的简单结论，对于长度 L 的文档，每隔 N 个汉字取一个 Shingle，这样一共取到 $L-N+1$ 个 Shingle，可见 N 的取值对效率和效果的影响都很大。很显然 N 最小取 2，最大取 L。

在特征抽取完毕后，就需要进行特征比对，因此网页查重的第 2 步就是相似度计算和评价。

I-Match 算法抽取的特征只有一个，因此只需要为文档计算一个签名值，如果两个文档的签名值相同，则文档相似。大规模文档做查重只需要维护一个哈希表即可，每来一个文档查找一次哈希表。如果哈希表的槽位（bucket）被置位，说明已经存在相似文档。

Shingle 算法抽取的特征有多个，因此处理起来稍微麻烦一些，比较的方法是记录完全一致的 Shingle 个数。然后除以两个文档的 Shingle 总个数减去一致的 Shingle

个数，这种方法计算出的数值称为“Jaccard 系数”，它可以用来判断集合的相似度。Jaccard 系数的计算方法为集合的交集除以集合的并集，即若令集合 $\boldsymbol{A}$ 与集合 $\boldsymbol{B}$ 的 jaccard 系数为 J，则有：

$$J = \frac{\boldsymbol{A} \cap \boldsymbol{B}}{\boldsymbol{A} \cup \boldsymbol{B}}$$

在图 3-14 所示的例子中，完全一致的 Shingle 分别是“中国足球队”、“世界杯决赛”、“界杯决赛阶”和“杯决赛阶段”。其中第 1 个文档具有 Shingle 总数是 30 个（计算方法为正文总长度-Shingle 长度+1），第 2 个文档 Shingle 总数为 22 个，因此两个文档的相似度（jaccard 系数）为 $\frac{3}{30+22-3} = 0.06$。在通过 jaccard 系数量化了相似度后，即可通过评价这个数值是否达到相似所需要的标准来判断是否相似。如果在搜索引擎中相似度达到 0.2 才能判断为相似，那么这个例子的相似度没有达到标准，因而不能看做是相似文档。

对比一下这两种方法，I-Match 算法提取特征时需要文本分词和词频比较的代价，因此提取特征比较复杂，但文档是否相似的计算简单；Shingle 算法提取特征简单，但文档是否相似的计算复杂，因此各有利弊。由于 Shingle 算法在性能上的优异表现，因而被广泛采用。

这里介绍的是最经典的两种方法，其他网页查重的方法还有很多。文献[李晓明 2004]中介绍了多种网页查重的方法，有兴趣的读者可以深入阅读。

网页查重的目标是消重，因此网页查重的第 3 步就是网页消重。

消重就不可避免遇到这样一个问题，即在相同或者相似的网页集合中保留哪一个，而消除哪些呢？从版权的角度考虑，应该尊重原创，过滤转载或者复制的网页；从网页寿命的角度考虑，过滤掉那些网站质量不高的网页，保留大型网站的网页；从容易实现的角度考虑，首先保留被爬虫抓取的网页，然后丢弃被抓取的相同或相似网页。最后一种方法最为简单实用，由于保留先被爬虫抓取的网页同时很大程度上也保证了优先保留原创网页的原则，因此被广泛采用。

综上所述，一般说来，网页查重至少需要如下 3 个主要步骤：

（1）特征抽取。

（2）相似度计算、评价是否相似。

（3）消重。

网页查重工作是分析系统不可缺少的重要一环，由于去除了重复网页，所以在接下来的其他模块的处理环节中，节约 PageRank 的计算代价、节省了索引存储空间、减少了查询成本、提高了查询结果的多样化效果。

3.4　中文分词

网页查重的工作完成后，分析系统在将分析的结果发往索引系统前还需要对正文进行分词，也称为“切词”（word segment）。分词的方法繁多，本节介绍一些基本方法。

3.4.1　什么是中文分词

任何文档都可以看做是一些连续的词的集合，然而中文并没有明显的词间分隔，这一点和英文不同。当然英文也有难点，例如时态和词性的变化等。在中文语法中，词汇是由两个或两个以上汉字组成的，并且句子是连续书写的，句子间由标点分隔。这就要求在自动分析中文文本前，首先将整句切割成小的词汇单元，即中文分词。

举个简单的例子来说明分词的难度，对于“学历史学好”这个句子，作为人来说，能够很容易得到正确的切分方法，即“学/历史学/好”（本书用“/”表示一个分词切分符号）。然而计算机要具有这样的智慧还有很多工作要做，否则可能分为“学历/史学/好”。

目前的分词手段主要依靠了字典和统计学的方法。由于索引是按照关键词建索引的，所以分词的效果直接决定了索引词及检索的效果。因此例如将文档“学历史学好”错分成“学历/史学/好”，所以索引时，只会对“学历”、“史学”及“好”这 3 个索引词建立它们与该文档的关联关系。这样用户查询“历史学”这个关键词时，就无法检索出这个文档，可见分词质量在很大程度上影响了搜索的结果和效果。

3.4.2　通过字典实现分词

先人为我们沉淀了厚重的历史和灿烂的文化，而其中的代表就是我们日常使用

的文字。也许正是因为没有断词，所以它才显得那么奇妙和多变。虽然这种多变在表达上提供了灵活性，但给搜索引擎带来了难题。在中文分词中主要体现了如下 3 种难分类型。

（1）交集型歧义

在阿拉伯数字字串 *AJB* 中，若 *AJ*∈*D*、*JB*∈*D*、*A*∈*D* 且 *B*∈*D*，则 *AJB* 为交集型歧义字段。此时，*AJB* 有 *AJ*/*B* 和 *A*/*JB* 两种切分形式，其中 *J* 为交集字段。例如，“从小学”，这个词可能有多种切分方法。

对于“从小学电脑”，正确的切分为“从小/学/电脑”。

对于“从小学毕业”，正确的切分为“从/小学/毕业”。

（2）组合型歧义

在字串 *AB* 中，若 *AB*∈*D*、*A*∈*D* 且 *B*∈*D*，则 *AB* 为组合型歧义字段。此时，*AB* 有 *AB* 和 *A*/*B* 两种切分形式。

“中将”这个词可能有多种切分方法。

对于“美军中将竟公然说”，正确的切分为“美军/中将/竟公然说”。

对于“新建地铁中将禁止商业摊点”，正确的切分为“新建地铁/中/将/禁止商业摊点”。

（3）混合型歧义

同时包含交集型歧义和组合型歧义，则为混合型歧义。

对于“人才能”，可能切分为“人才/能”、“人/才能”和“人/才/能”。

目前解决这些问题的技术手段主要借助字典和统计学的方法共同完成。我们首先从字典开始讲起，梁南元教授[梁南元，1987]最早提出利用字典的方法分词。但如何让计算机读字典，如何让计算机在各种字典的词义中选择合理的分词却并不容易。

日常使用的字典通常收录那些意义明确、使用频率高且生活中约定俗成的词汇，对于搜索引擎来说，字典中收录的词汇相当少。但即便是如此少的词，如何很快并有效地存储字典，以至于在需要知道一个字符串包含多少个字典中出现的词时能够快速地实现进行过程，成为首先需要解决的问题。

字典通常采用前缀树或者后缀树的数据结构存储，什么是前缀树？我们来举个例子。

将字典做成一个前缀树结构的数据结构，如图 3-15 所示。

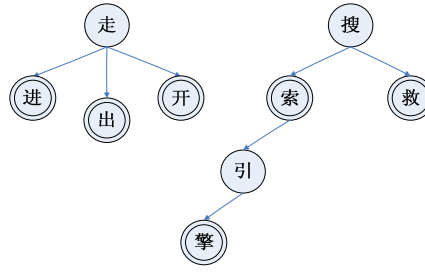


图 3-15　前缀树结构的词典组织形式

选择若干词建立一个前缀树数据结构作为示例，整个字典将是一个由多个后缀树构成的森林。其中从树顶开始，每一个字及其全部祖先构成一个词汇的一部分。对于图中双线圈的词，认为是一次终结，表示它及其全部祖先可以构成字典中的一个词汇。这里“走进”是一个词汇，而“搜索引”不是一个词，因为“引”是单线圈。“搜索引擎”是一个词汇，因为“擎”这个字是双线圈。这种结构十分类似于日常生活中字典的组织方式，即均以一个字开头，其后为以该字开头的全部词汇。日常生活中查字典的习惯也是如此，在字典中查一个词。首先查出该词汇的首个汉字在字典中的页数（这里可以理解为在字典森林中找到以某个字开头的前缀树），继而在其中查找相应的词汇（可以理解为在该词的后缀树中查到指定的词汇）。

分词的过程可以理解为一次查字典的过程，假定有一个句子“走进搜索引擎。”需要分词，那么计算机将会如何切分呢？

首先计算机读入“走”，找到“走”这个字典分支。然后读到“进”，查看“走”字打头的前缀树。发现“进”是双圈字，与“走”恰好构成一个词。继续读到“搜”时，发现“走”字开头的前缀树中没有继续的匹配，因此切分出“走进”这个词汇。

接下来找到“搜”这个字典分支，继而读入“索”。发现“索”是一个双圈字。系统确定了一个词“搜索”，然而还可能存在更长的匹配。此时系统继续读入“引”，由于这时“引”是单圈字，因此继续向下读入“擎”。发现“擎”为一个双圈字，由于下面继续读出的是句号，因此抽取出“搜索引擎”这个词汇。这样整个句子切分完毕，结果为“走进/搜索引擎”。

实际的分词并不总是如此简单，这种前缀优先的最大长度匹配的方法对于“学历史学好”这样的句子会分成“学历/史学/好”这样的错误结果，那么如何避免呢？

使用后缀树方法可以有效地降低这种错误，不过在从一个字典转到一个后缀树多少有些不够自然。后缀树的结构如图 3-16 所示。

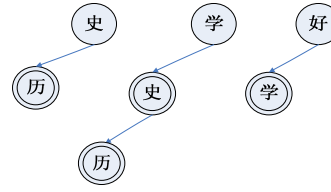


图 3-16　后缀树结构的词典组织形式

与前缀树的构造方法不同，树根表示一个词的结尾字，双圈字表示某个词的词头。我们知道，“历史学”、“历史”和“学好”是一个词，“学历史学好”被切分成“学/历史/学好”。

从句子的右边向左边读，首先找到“好”这个分支。然后读入“学”，恰好存在这样一个词，因此抽取出它。继续读入“史”，依次读入全部句子，直到切分结束，最后切分的结果为“学/历史/学好”。

虽然没有错分出“学历”，但是因为“好”这个字的特殊性，这样的逆向切分仍然不能达到理想的结果。

第 1 种方法称为“最大正向匹配法”（Maximum Matching Method），通常简称为“MM 法”；第 2 种方法称为“逆向最大匹配法”（Reverse Maximum Matching Method），通常简称为“RMM 法”。RMM 法的基本原理与 MM 法相同，不同的是分词的扫描方向。

这类分词的方法可以称为“贪婪算法”，“贪婪”在于总是认为最大匹配的词汇最优。或者说最大匹配可以使得切分出的词汇数量最少，而较少的索引词可以降低索引系统的工作量和最终索引文件的大小。

而事实上，贪婪并不总是最优。例如对于“学历史学好”，如果正向最大匹配，那么切分为“学历/史学/好”；如果逆向最大匹配，将会切分为“学/历史/学好”。无论如何切分都不理想，因此“贪婪”导致的局部最优很可能不是全局最优的问题被凸现出来。

不论 MM 法，还是 RMM 法都有不足。一个称为“N-Gram”的方法可以满足由于分错词而带来的损失，简单的如 2-Gram 方法，这种切分方法照顾了所有的可能。举个例子，“走进搜索引擎”按照 2-Gram 的切分方法，得到“走进”、“进搜”、“搜索”、“索引”和“引擎”。这有点类似于前面提到的 Shingle 方法，区别是 Shingle 方法由于是子文档集合。因此会在内部对相同的 Shingle 只保留一个 Shingle，而去掉其他相同的，从而保证集合的唯一性。

“N-Gram”的分词方法虽然有效地避免字典分词的错误而导致索引不完整的情况。但是分词是为了在索引阶段尽量减少索引项，而“N-Gram”的方法，特别是“2-gram”的分词法却会带来大量的关键词索引项，因为例如“进搜”这样的词都做成了索引项，这是极不经济的。

综上所述，没有一种分词方法能够解决全部的问题。虽然采用字典分词的方法作为主流的分词方法能够解决大部分的分词问题，但是字典总是滞后于语言的发展。很多新兴词汇需要每隔一段时间才能成批地收录进词典。因此能够及时地、自动地、准确地发现新词才能把字典分词的威力发挥到最大。新词发现主要通过统计推断的方法来实现。

3.4.3　基于统计的分词方法

虽然字典解决了分词的大部分问题，但是由于字典收录词数的限制，分词还需要具有新词发现的能力，即补充新发现的普遍被采用的流行的各种词汇到字典中。其中包括演艺明星姓名和网络流行语，甚至股票代码和火车车次都可能成为新词发现的目标。

那么如何借助一些先验的知识，来作出更加合理的分词方法呢？首先来了解一下 n-gram 模型。当给出这样一个填空，早起先刷__，我们都知道这个填空填什么最合适，但如果改成，要刷__，去掉了前缀“早起”，答案就变得很多。用概率的方法就是 P (牙|早起先刷)>P (牙|先刷)。从这一现象可以得到一个感性的认识，我们掌握的上下文越多，对句子的理解就会越准确。

下面我们通过句子的概率来给出分词的基本方法，任意一个句子 S 可以看做是一个顺序词构成的有序序列：$S=w_1w_2w_3\ldots w_n$，那么对于一个句子的概率，可以通过如下方法计算：

$$P(S)=P(w_1w_2w_3\ldots w_n)=P(w_1)*P(w_2|w_1)*P(w_3|w_1w_2)*\ldots*P(w_t|w_1w_2w_3\ldots w_{t-1})$$

那么我们刚刚给出的例句的概率为：

P(早起要刷牙)=P(早)∗P(起|早)∗P(要|早起)∗P(刷|早起先)∗P(牙|早起先刷)

这种计算句子概率的方法看上去科学，但随着句子的加长，训练预料的不足，概率的偏差会极大，因此通常采用 n-gram 的方法来计算句子概率，常见的有 unigram，bigram 和 trigram：

unigram： $P(S) = P(w_1) * P(w_2) *...* P(w_t) = \prod_{i=1}^{t} P(w_i)$

bigram： $P(S) = P(w_1) * P(w_2|w_1) *...* P(w_t|w_{t-1}) = P(w_1) * \prod_{i=2}^{t} P(w_i|w_{i-1})$

trigram： $P(S) = P(w_1) * P(w_2|w_1) * P(w_3|w_2 w_1)...* P(w_t|w_{t-1}w_{t-2})$
$= P(w_1) * P(w_2|w_1) \prod_{i=3}^{t} P(w_i|w_{i-1}w_{i-2})$

unigram 的模型下，计算句子的每个单词的概率，只考虑自身出现的概率，不考虑此前出现的句子概率，bigram 的模型往前看 1 个单词，也就是某一单词的概率是在前一个单词出现的上下文条件下，出现的条件概率。trigram 的模型一次类推。

做完前面的铺垫后再看分词，假定一个句子 $S=W_1W_2...W_n$,W_i 表示一个具体的单字，那么理论上该句子有多少中分法呢？分词的解空间是多大呢？假定 $F(n)$是 n 个词汇的全部分词方案数，那么 $F(n)=\binom{n-1}{0}+\binom{n-1}{1}+\binom{n-1}{2}+\cdots+\binom{n-1}{n-1}$。即分法包含了：一刀不切的方案数：$\binom{n-1}{1}$，只切 1 刀的方案数：$\binom{n-1}{1}$,...。这是一个二项展开式，可知 $F(n)=2^{n-1}$。因此分词的计算相当于在 2^{n-1} 中可能的选择中选择 1 个最优的，按照我们此前计算句子概率的计算方法找最大值，复杂度是一个 $O(n*2^{n-1})$，n 为计算最大值的代价，2^{n-1} 表示计算句子概率的代价，在句子较长时计算代价非常可观。

如何来降低计算的复杂性呢？一方面可以结合字典将明显不可能的可能性剪枝，将概率的对数看做是距离，概率大的距离就大，概率小的距离就小，分词就转化为求从第一个词汇到最后一个词汇的最大距离。以“早起先刷牙”为例，因为字典中不包含“先刷”这个词汇，因此“先刷”没有直达的路径，由于字典的帮助，减少了[早起][先刷][牙]和[早][起][先刷][牙]这两条路径的计算。最后，在结合字典的情况下，只要含这个字或词，就连线，得到如图 3-17 所示的路径走法。

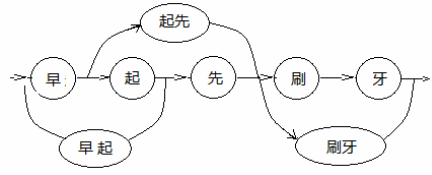


图 3-17　所有可能的分词切分法

如上图所示，不难得到从起点到终点的 5 种路径方案，它们的概率分别为：

$P(L1)=P(早)*P(起|早)*P(先|起)*P(刷|先)*P(牙|刷)$

$P(L2)= P(早)*P(起先|早)*P(刷|先)* P(牙|刷)$

$P(L3)=P(早)*P(起先|早)*P(刷牙|起先)$

$P(L4)=P(早起)*P(先|早起)* P(刷|先)*P(牙|刷)$

$P(L5)= P(早起)*P(先|早起)*P(刷牙|先)$

距离的累加通常直觉是加法，其实只要对 $P(L5)>P(L4)$ 两边取对数：$\log(P(L5))>\log(P(L4))$，即：$\log(P(早起))+\log(P(先|早起))+\log(P(刷牙|先))>\log(P(早起))+\log(P(先|早起))+\log(P(刷|先))+\log(P(牙|刷))$，就转化为对数的加法形式，和距离的概念就比较一致了。接下来，通过以往语料库的数据，选择 $L5$ 的这种概率值最大的切分方案。在求解过程中可以使用动态规划的方法降低计算的时间。

这种 n-gram 统计模型在实际使用时还会有很多问题，需要在训练语料不足，模式训练不充分的情况下，对 0 值进行 Good-Tuning，katz's backing-off 等方法进行平滑处理[Stanley 1996] [Chruch 1990]。除了分词，n-gram 统计语言模型在很多其他的场合都有应用，微软的自动对联机也以此来生成看上去更通顺的下联[Long 2008]。

3.5　PageRank

网页搜索的本质是网页信息的聚合，把本来很难聚合在一起的网页通过共同包含的关键词聚合起来。网页被聚合后就自然会产生排序问题。归纳起来就是既不能不排，也不能乱排，需要通过科学有效的方法将“好”的搜索结果按照“好”的程度依次排列。当然“好”是一个相当主观的且难以量化的概念，因此如何评价这个

“好”成为搜索质量的关键。在搜索引擎的发展史中，Google 发明的 PageRank 无疑是浓墨重彩的一笔。

3.5.1 PageRank 的来由

Google 公司的两位创始人 L. Page 及 S. Brin 在 1998 年提出了 PageRank 的概念[Page, et al.,1998]。在他们提出这个概念时，一方面，万维网的发展正处于信息大爆炸的时期。当时估计大约有 1 亿 5 千万网页，而且不到一年的时间网页的数目就会翻倍；另一方面，网页质量参差不齐，例如文献[Page, et al.,1998]中提到的网页信息“从 Joe 在哪里吃午饭到信息检索的期刊”无所不包，可以说实际有意义的、有价值的，以及经常被用户检索的网页规模并没有想象中的那么大，一个基本的方法是通过为网页排名使得重要性高且有价值的网页能够被优先检索。PageRank（网页排名）就是在这样的应用背景下诞生的。

3.5.2 PageRank 的基本想法

人们在万维网上冲浪的过程中，常常从一个起始网页开始（例如新闻门户网站），之后被那些带有链接文字（锚文本）所描述的网页吸引，从而点击并打开另一个网页。并依次往复，直到对冲浪感觉厌倦，而正是由于网页和网页通过链接关系构成的这个网络使得网上冲浪成为可能。然而直到 Google 提出了 PageRank 模型，这种网页间彼此链接关系的价值才被挖掘出来，这种挖掘过程也称为“链接分析”（link analyze）。简单地说，如果网页 A 链接到 B，那么表示网页 A 的编写者（网页工程师或者其他网页创造者）对网页 B 的一种认可。或者说网页 A 为网页 B 投了一票，网页 B 的重要性被网页 A 认可。

直觉上说，如下 3 点可以认为是网页重要性的一种评价。

（1）认可度越高的网页越重要，即反向链接（backlink）越多的网页越重要。

（2）反向链接的源网页质量越高，被这些高质量网页的链接指向的网页越重要。

（3）链接数越少的网页越重要。

举个例子，假定在一次比赛中一个网球选手 A 被另一个网球选手 B 击败。定义一个 A 指向 B 的链接表示这种胜负关系。即 $A \rightarrow B$，表示 A 输给 B，或者说 A 认可了 B 的厉害。那么很显然，对照上面得出下面结论。

（1）如果 B 的反向链接越多，即认可 B 厉害的人越多，因此 B 的排名越高。

（2）如果输给 B 的选手的排名越高，即认可 B 厉害的人也是厉害的人，那么 B 的排名越高。

（3）如果 B 输给的选手越少，即 B 认可的厉害的人越少，那么 B 的排名越高。

综上所述，赢的次数多、赢得对手质量高且输的少的选手的排名高是很自然的，PageRank 的算法的基本思想也来自于此。

在描述下载系统时曾提到过“随机冲浪”模型[Bernardo A. Huberman, et al，1998]，它是 PageRank 的理论基础，该模型描述网民访问网页的如下行为。

（1）用户随机选择一个网页作为上网的起始网页。

（2）看完这个网页后从该网页内所含的超链接随机选择一个页面继续浏览。

（3）沿着超链接重复浏览，直到对某个主题感到厌倦而重新随机选择另一个网页浏览。如此反复，直到结束。

3.5.3　PageRank 的计算公式

图 3-18 为文献[Page, et al.,1998]的原图。图中我们看到一个直观并简单的计算方法。以最左上角的网页为例，这个网页被 3 个箭头指向，假定共计得到了 100 分。然后通过两个链接（小横线表示网页内部的链接，方块表示一个网页）平均将这 100 分均分给了它所指向的两个网页。平均的意义在于假定用户是随机冲浪的，也就是点击第 1 个链接和第 2 个链接的可能性相等。这种由于指向关系而得到一定的分数，最后通过计算每个网页得到分数的多寡评价网页重要性的方法，就是 PageRank 的基本设计想法，也称作链接关系分析（link analyze）。

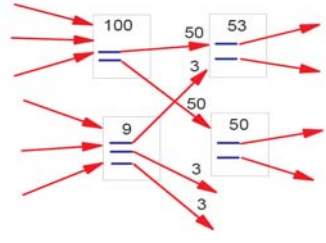


图 3-18　文献[Page, et al.,1998]的原图

如果分数被这样不断地传递下去，因为万维网彼此相连，这样的计算会没完没了吗？或者说 PageRank 的计算会收敛吗？

图 3-19 所示为对三个网页分别计算它们 PageRank 的一个迭代收敛的例子。

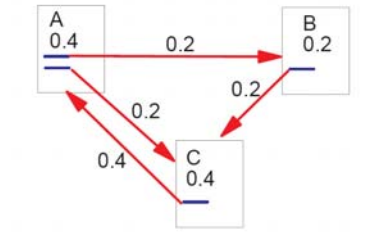


图 3-19　PageRank 的迭代收敛的例子

在图 3-19 中，无论怎样迭代的计算，不难发现，每个网页的得分都是固定不变的。网页 A 的分数平均分给了网页 B 和网页 C，网页 B 又完全给了 C，最后网页 C 把得到的分数又还给了网页 A。可以看出这 3 个网页得到的分数之和总是 1，而且无论如何循环计算，这种分数分布不再发生变化。这种特性在数学上称为“不变分布”，这也是 PageRank 计算收敛的最终奥秘。当然 PageRank 实际的计算公式考虑了各种因素，使得这种不变分布必然存在。这些数学上的原理，请读者参阅相关随机过程的书籍或论文予以透彻了解，下面我们将介绍 PageRank 的计算公式。

PageRank 采用以下公式计算：[Sergey Brin 1998]

$$PR_n(A) = (1-d) + d \times (\sum_{i=1}^{m} \frac{PR_{n-1}(T_i)}{C(T_i)})$$

说明如下：

（1）$PR_n(A)$：网页 A 的 PageRank 值。

（2）$PR_{n-1}(T_i)$：网页 T_i 存在指向 A 的链接，并且网页 T_i 在上一次迭代时的 PageRank 值。

（3）$C(T_i)$：网页 T_i 的外链数量。

（4）d：阻尼系数，$0<d<1$，$d \times \frac{PR_{n-1}(T_i)}{C(T_i)}$ 表示在随机冲浪模型中网页 T_i 将自身 d 的份额的 PageRank 值平均分给每个外链。由于网页 T_i 指向网页 A，因此网页 A

获得来自网页 T_i 的 C（T_i）分之一的 PageRank 值。

从整体上来看这个公式，一个网页 A 的 PageRank 值，由两方面得分决定，其比例分别为 1-d 和 d。

一方面，任何网页都有可能跳转到网页 A 来（不通过链接方式）。假定全集为 n 个网页，那么其中任意一个网页就有 n 个可能的无条件跳转的去处。每种跳转的概率相等，均为 1/n。因此网页 A 得到了每个网页的 1/n 的(1−d)的分数，每个网页的分数初始值都为 1。每次分数不论如何流动，其总值都为 n。因此，$\sum_{i=1}^{n}(1-d)PR_n(T_i)/n=1-d$ 的分数被分配到了网页 A 上，PageRank 中 1-d 的分数也可以理解为一个网页的基本分数，这是在整个计算中固定不变的常数。

另一方面，由于有些网页存在明确的指向网页 A 的链接，因此 $d\times(\sum_{i=1}^{m}\frac{PR_{n-1}(T_i)}{C(T_i)})$ 的分数被分配给了网页 A。

因此对于公式：

$$PR_n(A)=(1-d)+d\times(\sum_{i=1}^{m}\frac{PR_{n-1}(T_i)}{C(T_i)})$$

第 1 个部分可以认为是固有得分；第 2 个部分可以认为是因为被其他网页指向而得到其他网页的一部分得分。

在数学上可以证明，PageRank 由两方面因素共同决定的计算方法可以避免 Dangling page 和 Page sink 的问题[Langville et al 2003]。

PageRank 的计算涉及随机过程方面的数学原理，接下来通过一系列的例子来深入理解。假定存在如图 3-20 所示的简单的网页链接关系。

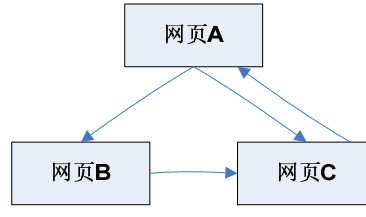


图 3-20　简单的网页链接关系

由 PageRank 的计算方法得到下列方程组，假定 d=0.5：

$PR(A)=0.5 + 0.5\times PR(C)$

$PR(B)=0.5 + 0.5\times (PR(A)/2)$

$PR(C)=0.5 + 0.5\times (PR(A)/2+PR(B))$

解这个三元方程组，得到：

$PR(A)=14/13 = 1.076\ 9$

$PR(B)=10/13 = 0.769\ 23$

$PR(C)=15/13 = 1.153\ 8$

从最后的 PageRank 值可以清楚地看出，在这 3 个网页中，网页 C 的重要性最高，它的得分为 1.1538 分；其次是网页 A；最后是网页 B。这种通过链接关系的分析得到网页重要性是合理的，网页 C 在这个局部是更被“认可”的，它被网页 A 和网页 B 指向；网页 A 被更高级别的网页 C“认可”，而网页 B 被网页 A“认可”，这样网页 A 和网页 B 同样具有一个 Backlink。由于网页 C 的级别大于网页 A，因此最终网页 A 的重要性大于网页 B。

此外，PageRank 还可以通过迭代计算的方法得到，计算过程如表 3-2 所示。

表 3-2　PageRank 迭代计算过程

迭代次数	*PR*（*A*）	*PR*（*B*）	*PR*（*C*）
0	1	1	1
1	1	0.75	1.25
2	1.125	0.75	1.125
3	1.0625	0.78125	1.15625
4	1.07812	0.765625	1.15625
5	1.07812	0.769531	1.15234
6	1.07617	0.769531	1.1543
7	1.07715	0.769043	1.15381
8	1.0769	0.769287	1.15381
9	1.0769	0.769226	1.15387
10	1.07693	0.769226	1.15384

这里初始向量为 $\boldsymbol{s}=\begin{bmatrix}1 & 1 & 1\end{bmatrix}^T$，表示每个网页的初始 PageRank 值均为 1。到达第 7 次迭代时，迭代结果和第 6 次迭代的结果基本相同，因此迭代停止。继续计算下去，PageRank 值不再变化或仅有很小的变化，这种稳定的 PageRank 值被用于衡量网页的重要性。然而当网页数量很大，这种链接关系复杂时，采用这些方法是低效的。

3.5.4　PageRank 的计算方法

在实践中，采用幂法（Power method）来计算 PageRank。

PageRank 的这个经典公式：

$$PR_n(A)=(1-d)+d\times(\sum_{i=1}^{m}\frac{PR_{n-1}(T_i)}{C(T_i)})$$

可以转化为求解 $\lim\limits_{n\to\infty}A^n x$ 的值，其中矩阵为 $\boldsymbol{A}=d\times\boldsymbol{P}+(1-d)\times ee^T/m$（$\boldsymbol{A}$ 也被称为“Google matrix”）。d 为阻尼系数，$\boldsymbol{P}$ 为概率转移矩阵，e^T 为 n 维的全 1 行，m 为全部网页个数（概率转移矩阵大小，x 是各个网页的初始 PageRank 值，初始每个网页的 PageRank 值均为 1。因此，ee^T/m 矩阵在每次迭代时与全部网页的 PageRank 向量乘积总是保持一个 n 维的全 1−d 的向量（在下面的例子中会提到）。这可以理解为总能够从其他网页得到 1−d 的分数，幂法计算过程的伪码如下。

（1）x ← 任意一个初始向量

（2）r ← $A\,x$

（3）if ($\|x\text{-}r\|$)$<\varepsilon$，返回 r

（4）else x ← r, goto 2

使用图 3-19 的例子来理解上面的计算过程。

（1）$\boldsymbol{P}$ 概率转移矩阵的计算过程。

初始化一个矩阵 $\boldsymbol{P}'$，若网页 i 存在一个指向网页 j 的链接，则 $\boldsymbol{P}_{ij}$=1；否则=0，

在图 3-19 的例子中，$\boldsymbol{P}'=\begin{bmatrix}0&1&1\\0&0&1\\1&0&0\end{bmatrix}$，然后将每一行除以该行非零数字之和。例如矩阵 $\boldsymbol{P}'$ 的第 1 行除以该行数字之和等于$\begin{bmatrix}0&1/2&1/2\end{bmatrix}$，每一行均按照这样的方法处理得到如下矩阵：

$$\boldsymbol{P}'=\begin{bmatrix}0&1/2&1/2\\0&0&1\\1&0&0\end{bmatrix}$$

这样的矩阵也称为“马尔可夫转移矩阵”（markov transmit matrix），它记录了每个网页跳转到其他网页的概率。对于第 1 行，网页 A 跳转到网页 B 和网页 C 的概率各为 1/2。通常使用其转置矩阵进行计算，即：

$$\boldsymbol{P}'^{\mathrm{T}}=\begin{bmatrix}0&0&1\\1/2&0&0\\1/2&1&0\end{bmatrix}$$

$\boldsymbol{P}'^{\mathrm{T}}$ 即

$$\boldsymbol{A}=d\times\boldsymbol{P}+(1-d)\times ee^{T}/m$$

公式中的概率转移矩阵 $\boldsymbol{P}$。

（2）$\boldsymbol{A}$ 矩阵计算过程。

由上面的计算结果，$\boldsymbol{P}=\begin{bmatrix}0&0&1\\1/2&0&0\\1/2&1&0\end{bmatrix}$，且易知 $ee^{T}/m=\begin{bmatrix}1/3&1/3&1/3\\1/3&1/3&1/3\\1/3&1/3&1/3\end{bmatrix}$，不妨设 d=0.5，这样得到 $\boldsymbol{A}=\begin{bmatrix}1/6&1/6&2/3\\5/12&1/6&1/6\\5/12&2/3&1/6\end{bmatrix}$，初始每个网页的 PageRank 值均为 1，即 x^{T}=(1,1, 1)。

（3）循环迭代计算 PageRank 的过程。

第 1 步

$$\boldsymbol{A}x=\begin{bmatrix}1/6 & 1/6 & 2/3\\ 5/12 & 1/6 & 1/6\\ 5/12 & 2/3 & 1/6\end{bmatrix}*\begin{bmatrix}1\\1\\1\end{bmatrix}=\begin{bmatrix}1/6+1/6+2/3\\ 5/12+1/6+1/6\\ 5/12+2/3+1/6\end{bmatrix}=\begin{bmatrix}1\\0.75\\1.25\end{bmatrix}=\mathrm{r}$$

由于 x 与 r 的差别较大，所以 r 赋值给 x，继续计算。

第 2 步 $\boldsymbol{A}x=\begin{bmatrix}1/6 & 1/6 & 2/3\\ 5/12 & 1/6 & 1/6\\ 5/12 & 2/3 & 1/6\end{bmatrix}*\begin{bmatrix}1\\0.75\\1.25\end{bmatrix}=\begin{bmatrix}1.125\\0.75\\1.125\end{bmatrix}$，反复迭代，最终的计算结果依次为 $\begin{bmatrix}1.062\ 5\\0.781\ 25\\1.156\ 25\end{bmatrix},\begin{bmatrix}1.078\ 125\\0.765\ 625\\1.156\ 25\end{bmatrix},\begin{bmatrix}1.078125\\0.769531\\1.152344\end{bmatrix},\ldots,\begin{bmatrix}1.076\ 9\\0.769\ 23\\1.153\ 8\end{bmatrix}$ 和 $\begin{bmatrix}1.076\ 9\\0.769\ 23\\1.153\ 8\end{bmatrix}$。

在最后两次的结果近似或相同的情况下迭代结束，这个结果和解方程、简单迭代的方法求出的结果是一致的。用幂法计算 PageRank 总是收敛的，即总会在有限的迭代计算轮次内结束。

参考文献[Langville et al 2003] [Haveliwala et al 2003] [Lars Elden 2003]可以进一步深入理解 PageRank 计算过程收敛的奥秘，以及参数 d 是如何影响收敛速度的基本原理。

在实际应用中，计算 PageRank 的过程要比想象的更为复杂。例如从提高收敛效率角度，加快计算速度；采用将 Google matrix（PageRank 这个概率转移矩阵学术上也称作 Google matrix）分块的思想执行并行计算减少磁盘 I/O。从而发挥多 CPU 的作用，有兴趣的读者可以进一步探索。

自从 PageRank 的计算方法被公开后，各种为了提高排名而进行的作弊行为也随之而来。Google 及其他采用类似技术的搜索引擎公司都面临着前所未有的压力，PageRank 的科学性和可信度也受到了极大怀疑。但是科学家们采用了各种反作弊的技术手段，沉重打击了作弊者的不道德行为，维护了 PageRank 应有的价值。

至此，我们结束了分析系统的全部旅程，为了更好地从全局理解分析系统各个模块的工作关系，下一节中将和大家一起从宏观全局的视角重新回顾分析系统。

3.6 分析系统结构图

分析系统在搜索引擎的架构中承担了网页结构化、网页消重、文本分词及 PageRank 计算等 4 项基本任务。通过前面的分块学习，最后通过一个分析系统结构图来全面了解分析系统的运作方式。分析系统结构图如图 3-21 所示。

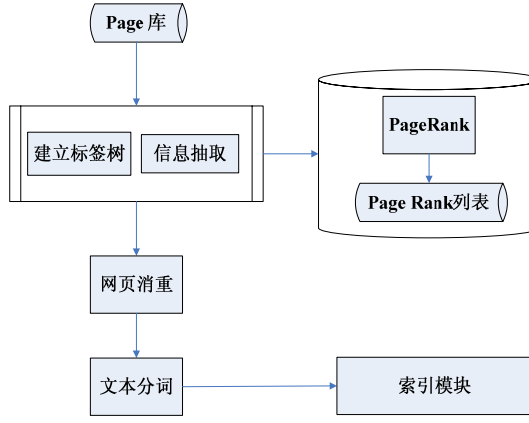


图 3-21 分析系统结构图

其中 Page 库是下载系统通过爬虫下载到的原始网页，分析系统通过以下步骤完成对这些网页的分析工作。

（1）经过一个网页结构化的过程，包括建立标签树并从网页中抽取有价值的属性，完成从原始网页打包成一个网页对象的过程。

（2）网页消重模块丢弃冗余的页面，相似或相同的网页仅保留一个传给分词模块。

（3）文本分词将正文切分成以词汇为单位的集合。

（4）将分析的结果发往索引模块，进行索引入库。

以上 4 个步骤中网页结构化、消重、分词这三项工作同步计算，因此速度非常快。仅 PageRank 的计算非常耗时，而且必须积累一定数据后才能生成一次数据（这里信息抽取过程中得到的网页链接信息发往 PageRank 计算服务器）。由于执行一次 PageRank 的计算代价极大，因此采用离线计算方法。离线计算的结果是一个 PageRank 列表，其中包含每个网页的一个 PageRank 值。该值越高，网页的重要性

越高，在检索时就越容易被检索到。在索引系统中会利用该 PageRank 值对文档列表索引项的先后顺序产生影响，也就是越是重要的网页（PageRank 值高）在索引中越能够占据有利的位置。

回顾本章，网页分析系统的工作层层深入。除了第 1 步分析工作是为本层服务，其余的分析工作包括网页消重、分词及 PageRank 的计算都是为了索引系统查询系统服务，因此只有在深入了解索引系统和查询系统的原理后我们才能深刻理解这些分析系统工作的重大意义。接下来我们将继续走进搜索引擎的索引系统中，领略索引系统的魅力。

参考文献

[Abdur chowdhury et al 2002]Collection Statistics for Fast Duplicate Document Detection ABDUR CHOWDHURY, OPHIR FRIEDER, DAVID GROSSMAN, and MARY CATHERINE McCABE Illinois Institute of Technology ACM Transactions on Information Systems, Vol. 20, No. 2, April 2002, Pages 171–191.

[Bernardo A. Huberman, et al，1998]　Bernardo A H,Peter L T P, James E, et al. Strong regularities in world wide web surfing. Science,1998.

[Broder 1997]A. Broder, "On the Resemblance and Containment of Documents," *sequences*, p. 21-29,　Compression and Complexity of Sequences 1997,　1997.

[C.-N.Hsu, et al.,1998] C.-N. Hsu and M.-T. Dung. Generating finite-state transducers for semi-structured data extraction from the web. Information Systems, 23（8）:521-538，1998.

[Chruch 1990]Church, K., and Gale, W. (1990) , Poor Estimates of Context are Worse than None,Third Darpa Workshop on Speech and Natural Language, Hidden Valley, PA.

[Haveliwala et al 2003] Haveliwala, T.H. and Kamvar, S.D.. The second eigenvalue of google matrix. Stanford University Technical Report，2003.

[Hammer,et al.,1997] J. Hammer, H. Garcia-Molina, J. Cho, A. Crespo, and R. Aranha. Extracting semistructured information from the web. In Proceedings ofthe

Workshop on Management of Semistructured Data, pages 18-25，May 1997.

[J. Hammer et al.,1997]J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo Extracting Semistructured Information from the Web.

[Lars Elden 2003] Lars Elden, The Eigenvalues of the Google Matrix Technical Report LiTH-MAT-R-04-01.

[Langville et al 2003] Langville, A.N. and Meyer, C.D. Deeper inside pagerank. Technical Report, NCSU Center for RES SCI Comp. 2003.

[Long 2008] Long jia,Ming Zhou,Generating Chinese couplets using a statistical MT approach,Proceedings of the 22nd International Conference on Computational Linguistics(Coling 2008),pages377-384 Manchester,August 2008.

[Page, et al.,1998] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Digital Library Technologies Project 1998.

[S.-H.Lin et al,.2002] S.-H. Lin and J.-M. Ho. Discovering informative content blocks from web documents. SIGKDD，2002.

[S. Brin 1998] S. Brin and L. Page. The anatomy of a large-scale hyper textual Web search engine. Computer Networks and ISDN Systems, 30(1-7):107-117，1998.

[Sergey Brin et al 1995]Sergey Brin , James Davis , Héctor García-Molina, Copy detection mechanisms for digital documents, Proceedings of the 1995 ACM SIGMOD international conference on Management of data, p.398-409, May 22-25, 1995, San Jose, California, United States.

[Stanley 1996]Stanley F. Chen, Joshua Goodman (1996, Harvard University), An Empirical Study of Smoothing Techniques for Language Modeling, Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics.

[梁南元，1987] 梁南元，“书面汉语自动分词系统——CDWS”中文信息学报 1987 年 第 02 期.

[李晓明 2004] 李晓明，闫宏飞，王继民，“搜索引擎——原理、技术与系统”，科学出版社.

第 4 章　搜索引擎的索引系统

4.1 知识准备

在搜索引擎的 5 大系统中，第 3 个系统称为“索引系统”。该系统就好像搜索引擎的数据大本营，在这里存储并索引了数以亿计的网页。在搜索引擎早期的发展中，能够索引的网页数量代表了整个行业的技术发展水平。由于需要支持多用户同时检索，索引系统还必须提供低于秒级的检索时间，因此“存得下”和“查得快”是围绕本节的重要话题。

与前几个章节一样，一起来做一下热身活动，了解一些基本概念。

4.1.1 信息

信息是能够被传达和理解的消息，是通过学习和经历获得的知识，是用来做出判断的一组事实[WordNet]，不同的角度上对信息具有不同的解释。这里我们认为信息就是结构化的网页数据，即一组有价值的数据的集合。

4.1.2 索引

索引也是一种信息，可以说是信息的信息，或者说是描述信息的信息。例如，书中包含的目录。其中每一条目就是一个索引，用来标识某个章节的页码。帮助读者快速浏览，索引就是这样一种短小精练的检索信息的信息。

4.1.3 倒排索引、倒排表、临时倒排文件、最终倒排文件

为了便于理解整个章节笔者做出如下定义：

倒排表是指存放在内存中的能够追加倒排记录的倒排索引。倒排表是迷你的倒排索引。

临时倒排文件是指存放在磁盘中，以文件的形式存储的不能够追加倒排记录的倒排索引。临时倒排文件是中等规模的倒排索引。

最终倒排文件是指由存放在磁盘中，以文件的形式存储的临时倒排文件归并得

到的倒排索引。最终倒排文件是较大规模的倒排索引。

倒排索引作为抽象概念，而倒排表、临时倒排文件、最终倒排文件是倒排索引的三种不同的表现形式。

4.1.4　其他概念

索引部分概念很多，因此本章 4.2~4.4 节分别介绍全文检索、文档编号、正排索引、倒排索引的基本概念。在集中理解索引系统的主要概念后，接下来再了解索引创建中的一些计算细节。

4.2　全文检索

全文检索（full-text retrieval）技术的出现是信息检索领域的一场革命，它细化了信息检索的粒度，提供了实现多角度，多侧面且全新的信息检索体验。因此搜索引擎全面采用了这种崭新的技术，并使之成为主流的检索方法。

早期的信息检索主要通过检索数据信息的外部特征，例如标题、作者、摘要、附录及资料的编号等。这样的检索系统常见于图书馆的馆藏图书检索中，它主要存在如下两个大问题。

（1）检索结果排序不理想。

（2）只能对标题进行检索。

出现这些问题是因为没有考虑到文档内容（本章使用文档笼统地代表书籍或者网页）。全文检索顾名思义，是对文档的全部信息进行检索，这些信息包括标题和正文等。简单地说，全文检索的内在本质归纳起来就是如下两条。

（1）文档的全部文字参与索引。

（2）检索结果能够提供检索词出现的实际位置。

在全文检索的过程中，只需要用户提供一个或多个检索关键词（以下简称“关键词”），不仅能够检索出命中的文本，还能够提供这些关键词在文本中出现的位置。受到搜索引擎检索结果的展示窗口的限制，不可能把全部关键词出现的位置一一列

出，最终的展示效果如图 4-1 所示。

图 4-1　关键词位置的展示方式

其中椭圆矩形表示网民向搜索引擎提交的关键字，矩形框表示关键字在文档中的位置。这里搜索引擎采用一种称为“标红”的技术用红色标出关键词，其余文字为黑体，这样的方法比用数字标出位置信息更加直观。接下来用户就可以通过上下文、关键词在文本中的位置，以及检索到的文档标题等因素确定哪一条才是符合需要的检索结果。例如图 4-1 是用“全文检索”这个关键字在 Google 搜索引擎中进行检索的结果。

当今搜索引擎无一例外地采用了全文检索，使得越来越多的人开始使用搜索引擎查找信息，信息检索的面貌也为之一新。在这革命性的搜索体验背后蕴涵的是大量的信息组织的智慧，它最终使得全文检索从梦想成为现实。

4.3　文档编号

一个唯一的编号是描述一个复杂事物最好的方法，每一个文档赋予一个唯一的编号后，这个编号就能代表这个文档。

4.3.1　编号的本质

编号的历史能够追溯到古老的时代。部队有番号、单位职工有工作证号、大中专院校学生有学号，以及公民有身份证号等。从各种各样的编号中不难发现这样的

规律，即凡是需要区别大量个体的情况就会用到编号。一个家庭成员之间不会用编号来区分，因为用姓名更加自然并亲切。然而，如果某个家庭成员来到学校读书，由于可能存在同名同姓的同学，因此采用姓名就不能有效地区分，使用学号就成为必然的手段。这样通过一定的编码方式可以保证某个年级某个院系某个班级的某个学生拥有唯一标识自身的学号。

对于搜索引擎来说，编号不仅仅是保证唯一，而且它好像是一道连通信息和索引的桥梁。搜索引擎需要处理的信息是海量的，为了能够快速检索到这些信息，通常的做法是为信息创建索引。索引也是信息，是那种能够用来找到信息的信息。对于搜索引擎来说，网页是信息，关键词就是索引，索引相对于信息来说是轻量级的。文档千差万别，而关键词总是有限的。因此虽然处理的是海量信息，但是转化到索引后在空间占用上就会变得相当小。检索在索引上完成后，得到一组匹配关键词的文档编号。继而将这些文档编号在文档信息库中取出，通过一系列计算展示出来，编号就是这样在检索中发挥了重要的作用。

一个网页被爬虫抓取，经过分析系统的分析。然后得到结构化的网页对象，为此首先需要为结构化的网页对象进行编号存档。在索引系统中，我们统一将这种结构化的网页对象（包含标题、正文和 URL 等信息的结构体）称为“文档”，网页的编号称为“文档编号”。

文档编号和日常生活中的编号最大的不同在于，它不需要具有具体的含义。与机器不同，日常生活中使用编号大多具有某种意义。例如我们的身份证号每一段都有其特定的含义，然而文档编号有其特殊性。在信息索引阶段，编号无处不在。各种检索及排序计算都依赖于编号，因此文档的编号方法会有一些额外的特别要求。

4.3.2　文档编号的方法

文档编号需要满足下面 3 个条件。

（1）任何一个文档在其生命周期中仅有一个编号。

（2）任何两个不同的文档的编号不同。

（3）编号在计算中尽可能高效，并且为了便于存储，要越短越好。

第 1 个条件表明文档编号和文档内容无关，有些文档内容常常发生变化。如果

编号与内容有关，编号的稳定性将不能够被满足。

编号不能直接由文档文件名代替，因为很多网站都有类似 index.html 和 welcome.html 等这样通用的文件名。如果编号由文件名代替，那么就破坏了第 2 个条件，即不同的文档由于相同的文件名而赋予相同的编号。

综上所述，文档内容和文件名均不能代表一个文档。事实上，只有 URL 才能够唯一地表示一个文档。通过浏览器输入一个 URL 只能到达唯一网页，这首先保证了第 2 个条件；其次也满足了第 1 个条件，网页的内容变化不会影响其 URL。因此，URL 是天然的计算文档编号的来源。

由于中文万维网网页规模将来必然达到百亿量级的规模，如何合理地对文档进行编号才能符合第 3 个条件要求的高效计算呢？

显然将文档的 URL 直接作为编号是不合适的，一个 ASCII 字符为一个字节，这样的一个 URL（http://sports.sohu.com/ 20070411/n249364 259.shtml）就需要多达 48 个字节。不论是计算还是存储都是不经济的。计算不经济主要体现在字符串运算慢，这是因为字符串编号在查找和排序时的代价都很大。排序和查找都是利用多次比较完成的，时间的耗费主要在比较操作上，字符串之间的比较耗时远大于整型的比较；存储不经济主要体现在字符串的空间占用较大，增大了磁盘 I/O 的次数。此外，空间占用量大还会影响计算效率。空间占用越大，内存工作集也就相应变大，内存和对换区的换入换出机会就增大。

综上所述，采用对字符串映射为占 64 比特大小的长整形整数，利用整数计算高效，存储代价小的特点进行文档编号是必然的选择。

整型编号不仅可以节约大量存储空间，而且大大提高检索计算中比较排序的效率。采用 URL 字符串签名的方法（参见第 2 章），一个 32 位整型能够表达 40 亿量级的数。对一个 URL 字符串签名得到一个占两个整型（8 个字节）大小的 MD5 签名，理论上它能够表达足够多的网页数量，这被认为符合当前时代万维网网页数量的编号要求。对于大规模的网页编号来说，两个整形的空间耗费还是太大。通过编码的技巧可以让其变得更短，更适合计算的需要。

4.3.3 游程编码

本节介绍一种通用的编码方式，称为“游程编码”，使用这种方式进行编号长度

压缩。在后面还会提到这个技巧，因此这里可以作为一个知识准备。

假定一个文档编号序列为 1，17，34，69，489，512，3456，这些都是文档的内部编号。编号为升序序列，即排在后面的数不小于排在前面的数。不妨假定编号为一个整型大小，并且能够满足 40 亿个不同个体的编号需要。上面这个文档编号序列包含 7 个整数，因此需要 7 个整型（28 个字节）的数组来存储。

对于这种单调的序列，可以将增量整数序列被变换为差分序列，这种编码方式也称为“差分编码”（gap encoding）。通过转换得到 1，17-1，34 -17，69-34，489-69，512-489，3456-512，即 1，16，17，35，420，23，2944。由于差分序列除第 1 个编号外其余都保存的是间隔距离，因此带来一个好处和一个坏处。

好处是因为存放的是相对数，所以除第 1 个数以外，序列中的数都变得较小。

坏处是为了取出序列中某个特定值，需要由头至尾取出所有数据，并进行多次加法运算。一旦缺失一块数据，则无法序列中的后续部分无法恢复。

举个例子，在 1，16，17，35，420，23，2944 这个序列中，如果需要取序列中第 7 个数，首先必须读出全部数据，进行连加，也就是需要计算 1 + 16 + 17 + 35 + 420 + 23 + 2944 = 3456。如果由于各种原因，中间的数据缺失或者出现异常，例如第 4 位应该存放 35 而读取出来为-35，这使得后续的数据无法恢复。通过这个例子不难看出，一方面付出了 CPU 的代价；一方面付出了维护数据完整性的保证，继续来看得到怎样的补偿。

由于序列中的数都变得很小，所以可以采用一种变长编码（Variable Byte Coding）[Williams 1999]的技巧。这是一种字节对齐的编码方式，即将整数转化成二进制后，以 7 位为单位对其分段。每段段尾加一位成为 8 位，恰好用一个字节表示。末位为 0 表示该段是最后一段，为 1 表示还有后续段。例如，135 的二进制编码表示为 10000111 通过编码后为 00000011 00001110。第 1 段（00000011）末尾的 1 表示该字节为段首（还有后续段），倒数第 1 个 1 表示 135 二进制数据中的第 1 个 1。第 2 段（00001110）末尾的 0 表示该段不是段首，其余 7 位表示 135 二进制数据中的后 7 位（0000111）。编码的过程很容易，解码的过程也相当快。

在实践中，Variable Byte Coding 编码不是压缩率最高的编码方式，例如其压缩率不如 γ code(Elias)，但由于 Variable Byte Coding 具有字节对齐的优秀特性，所以解码非常快速。具体的解码过程参考下述伪码。

解码过程的伪码为：

```
VariableByteRead（A）
int i = 0x1;
int v = 0;
while（（i & 0x1） == 0x1 ）
 i ← HEAD（A）;
 v ← （v<<7） +（（i>>1）&0x7F）;
return v;
```

在上述伪码中，HEAD(A)表示从 A 中读取头部的一个字节，并去掉这个字节。例如，135 表示为 00000011 00001110。

第 1 次执行 HEAD 操作返回 A 的段首字节，该字节的二进制表示为 00000011 并且去掉这个头部字节。因此此时 A 的二进制表示为 00001110，这时 *i* 保存段首字节（00000011）。对其右移一位（*i*>>1），取其字节后 7 位（&0x7F），此时 *i* 为 00000001。然后 *v* 初始化为 0 ，这样第 1 次执行后 *v*=1。

第 2 次执行 HEAD 时，返回的结果是字节（00001110），并存放到变量 *i* 中，即十进制的 7。将变量 *i* 右移动 1 位，并取后 7 位，得到 00000111，加上第 1 次计算的 *v* 右移 7 位，最后得到 10000111，即 135。

对于 32 位的整数，采用 Variable Byte Coding 编码方式。小于 128 的数可用 1 个字节来表示，从而节省了 3/4 的空间；而大于 268 435 455（二进制 1111111-1111111-1111111-1111111 共 24 位，每位存储 7 位，因此需要 4 个字节存储 268 435 455）的数则需要 5 个字节。这样大于 268 435 455 的数用 Variable Byte Coding 编码方式需要的存储空间超过了一个整形的大小，不但没有达到压缩的目的，反而增大了空间。但是由于经过排序，序列之间的差值都会保持相对较小，因此差值达到如此大的距离（大于 268 435 455）的概率极小。可以忽略这种情况带来的代价，整体的存储是经济的。此外，由于计算机以字节为操作单位，字节对齐的编码往往更能发挥硬件的优势，所以可变字节编码的压缩和解压缩的速度都较快。

这个差分序列中的 1，16，17，35，23 都小于 128，因此只需要 1 个字节。420 需要两个字节（420 的二进制表示为 110100100，每个字节只有 7 个有效位，因此这里需要两个字节）。同理，2 944 也需要两个字节。这样全部序列需要 9 个字节，和原序列需要 28 字节相比压缩了 60%。由于这些表示文档编号的序列存放在磁盘中，因此节约存储相当于减少了磁盘 IO 的次数。压缩存储是典型的以时间换空间，一方面由于多了编码和解码的过程，导致 CPU 的消耗增加；另一方面由于节约了磁盘，而减少了磁盘读取次数。一般来说，I/O 是索引系统的瓶颈。而且 CPU 和 I/O 之间

性能差距还在不断扩大，因此采用这种技巧带来的减少了磁盘 IO 的次数的好处完全可以抵偿因此带来的 CPU 计算量增大的不利结果。

文献[Witten 1994]堪称是信息检索的经典书籍，其中介绍大量关于大规模数据压缩的其他方法和技巧，它们各有优劣。需要进一步钻研的读者可以通过阅读本书全面地理解处理大规模数据的各种技巧，通过 Variable Byte Coding 的学习，充分理解排序对压缩的关键作用，排序带来的一些冗余数据和处理这种冗余数据的方法。

在解决了文档编号存档的问题后，接下来就是如何查找到这些文档。记住编号超出人类记忆的能力范围，记住标题也难度颇大。那么对于酒香还怕巷子深的问题，用什么方式来索引这些文档才能符合全文检索的需求呢？下一节我们将着重介绍有关倒排索引的概念。

4.4　倒排索引

全文检索的检索是通过关键词来进行检索，因此为关键词建立索引是很自然的。我们把按关键词创建的索引称为“倒排索引”，在这里关键词称为“索引词”，因为并不是所有的关键词都会创建索引。

4.4.1　经典的倒排索引

笼统地说，索引系统的索引如图 4-2 所示。

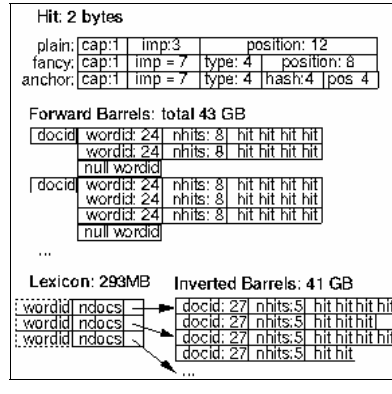


图 4-2　索引系统的索引

索引系统中的索引[S. Brin 1998]包含了 3 个概念，分别是命中（Hit）、正排索引（Forward Index）和倒排索引（Inverted Index）。Hit 表示索引词在文章中的位置（position）和字体等信息，这里为了突出重点，忽略 Hit 的影响，集中精力了解正排索引和倒排索引。

4.4.2 正排索引（前向索引）

正排索引也称为“前向索引”。它是创建倒排索引的基础，具有以下字段。

（1）LocalId 字段（表中简称“Lid”）：表示一个文档的局部编号。

（2）WordId 字段：表示文档分词后的编号，也可称为“索引词编号”。

（3）NHits 字段：表示某个索引词在文档中出现的次数。

（4）HitList 变长字段：表示某个索引词在文档中出现的位置，即相对于正文的偏移量。

由于一篇文章中的某些词可能出现多次，而且位置不同，而全文检索的本质要求是把这些位置标识出来，因此 HitList 中的每个命中都表示索引词在文档的某个位置中出现了一次，这个序列为单调递增序列。基于游程编码的方法，变升序序列为差分序列，采用前文提到的 Variable Byte Coding 方法编码可以大大压缩正排索引的 HitList 字段。

在正排索引中 LocalId 采用升序序列编号（假定编号采用自增 1 的方式递增），这为下面的计算创造条件。进行倒排索引的转化时，由于正排索引中 Lid 天然的有序性，因此在正排索引转化为倒排索引的创建过程中，自然可以保证倒排索引中每个词汇对应的文档编号也是有序的，倒排索引将在下一节中介绍。

这样，正排索引如图 4-3 所示。

LId	WordId	NHits	HitList
Doc1	Word1	m	Hit1,...Hitm
	Word2	n	Hit1,..Hitn
	...		
	WordN		
	NULL		
Doc2	...	...	...
	NULL		

图 4-3 正排索引

通过一个例子来了解正排索引的创建过程。假定存在这样一个编号为 1 的文档，其全文为“走进搜索引擎，学习搜索引擎”，分词的结果为“走进/搜索引擎/学习/搜索引擎”。不妨为“走进”编号为“T1”，“搜索引擎”编号为“T2”，“学习”编号为“T3”。通过计算得到“走进”出现 1 次，出现位置为 1；“搜索引擎”出现两次，出现位置为 3 和 9（图中存放的为未压缩的差分序列 3 和 6）；“学习”出现 1 次，出现位置为 7。创建的正排索引如图 4-4 所示。

LId	WordId	NHits	HitList
1	T1	1	1
	T2	2	3,6
	T3	1	7
	NULL		

图 4-4　正排索引示例

HitList 是变长的，因此需要 NHits 这个字段标记其长度，这样才能读出全部的正排索引数据。假定每个域都是一个字节大小，而 HitList 变长。在上面这个例子中，假定这个正排索引依次存放在一个数组 Array 中，则当读取到 Array[2]时，读取的内容为 T1 的 NHits，读出的结果为 1。由于 T1 出现了 1 次，因此在 T1 的 HitList 中仅存放了一个位置信息。这样在读取 Array[3]后，接下来读取 Array[4]则能够判断为下一个索引词 T2。正是由于 NHits 的这种表示长度的作用，全部的数据才能被有效地读取。

最后用 NULL 表示为一个结束符的这种设计是很巧妙的；否则，正排索引可能是这样，如图 4-5 所示。

LId	WordId	NHits	HitList
1	T1	1	1
1	T2	2	3,6
1	T3	1	7

图 4-5　冗余存放 DocId 的正排索引

在图 4-5 中，为了结构化数据的需要，在去掉表示结束符的 NULL 后，正排索引必须冗余地存放 DocId，因此一个标记结束符的字段有效地压缩了正排索引的大小。

本质上说，正排索引以文档编号为视角看待索引词，也就是通过文档编号去找索引词。任给一个文档编号，能够知道它包含了哪些索引词、这些索引词分别出现的次数，以及索引词出现的位置。然而全文索引是通过关键词来检索，而不是通过文档编号来检索，因此正排索引不能满足全文检索的要求。

虽然正排索引不能满足全文检索的需要，但是正排索引为创建倒排索引创造了有利条件，是计算倒排索引的不可缺少的一环。

4.4.3 倒排索引

倒排索引是一种以关键字和文档编号结合，并以关键字作为主键的索引结构。倒排索引分为两个部分。

（1）第 1 个部分：由不同索引词（index term）组成的索引表，称为“词典”（lexicon）。其中保存了各种中文词汇，以及这些词汇的一些统计信息（例如出现频率 nDocs），这些统计信息用于各种排名算法（Ranking Algorithm） [Salton 1989;Witten 1994]

（2）第 2 个部分：由每个索引词出现过的文档集合，以及命中位置等信息构成，也称为“记录表”（posting file）或“记录列表”（posting list）。如图 4-6 所示。

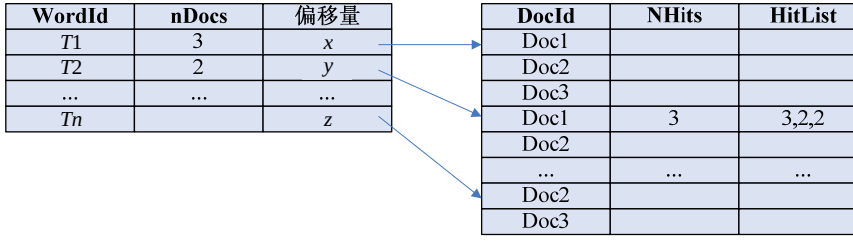


图 4-6 倒排索引

左边的表结构（词典）记录索引词 Id 号、匹配该索引词的文档数量，并匹配文档在记录文件内的偏移量，通过这个偏移量就可以读取记录文件对应区域的信息。例如在图 4-6 中，通过读取 T1 的偏移量 *x*，读取在记录文件中 T1 命中的相关文档 Doc1、Doc2 和 Doc3 的相关信息。

右边的表结构（记录表）记录文档编号（DocId）、索引词在该文档的命中个数（NHits），以及命中域的列表（HitList）。例如在图 4-6 中，记录文件显示了 T2 关键词在 Doc1 中出现了 3 次。

图 4-6 所示的倒排索引示例中，以索引词 T2 为例，T2 在两个文档中出现。通过偏移量在记录文件中找到了存放与 T2 有关的信息，即 T2 匹配的 Doc1 和 Doc2 两个文档，并且在 Doc1 中出现了 3 次，位置分别为 3、5 和 7（图中 3、2 和 2 表示为差分序列后的实际值，即 3、5-3 和 7-5）。

注意到这种按照索引词组织文档的方式，在索引词 WordId 一边，其 Id 号不会重复；而在 DocId 一边，由于每个文档都可能包含多个索引词，DocId 的重复非常普遍，因此对 DocId 就需要进行大规模的压缩。

压缩编码也采用 Variable Byte Coding 编码方法进行压缩，首先对 DocId 排序，接下来将 DocId 的递增序列为差分序列，最后用 Variable Byte Coding 编码方法进行压缩编码。例如这样的 DocId 序列（22，5，9，1），通过排序得到（1，5，9，22），变差分序列得到（1，4，4，13），压缩编码后得到（2，8，8，26），对于小于 128 的数进行压缩编码，相当于乘 2（详细参见本章第三节中的相关内容）。

最后，在倒排索引中，按照何种顺序存放 DocId 更加有利于检索呢？在图 4-6 中，T1 这个词有 Doc1、Doc2、Doc3 与之相匹配，也就是在这些文档中都出现了 T1，那么在倒排索引的记录表中，哪个文档编号先存放，哪个文档编号后存放呢？这种存放顺序的策略大致有如下 3 种。

（1）按照 DocId 升序存放。

（2）按照索引词在文档中出现次数降序存放。

（3）记录表分块存放，块内按 DocId 升序存放，块间按 PageRank 值降序存放。

对于方案 1 来说，它有助于在多关键词查询中，得到相同的 DocId（在第六章中介绍查询系统时，会提到因为 DocId 有序而为查询带来的好处）。并且能够对 DocId 进行压缩编码，同时降低磁盘 I/O 的开销。

对于方案 2 来说，按照索引词 Id 在文档中出现的次数降序排序，因为索引词出现次数多的文档与查询关键词的相关性越好（这里，索引词和关键词是同一个词，在索引系统中称为“索引词”，在查询系统中称为“关键词”或“查询词”），这是很自然的，检索就是检索哪些与查询词相关性高的文档。文献[S. Brin 1998]阐述了一种折中的设计方案，即不同的索引词命中区分对待，对于索引词在标题或者锚文本（Anchor）命中的文档，以及命中信息存放在一个记录表中，不妨称为“表 A”，表内数据按 DocId 排序；命中其他位置的文档及其命中信息存放在另一个记录表中，称为“表 B”，同样表内数据也是按照 DocId 排序。这样对于每个索引词的查询，优先在表 A 中查询。只有当结果数不够时，再到表 B 中查询。这样既照顾到了压缩存储的需要，也照顾到了相关性，通常标题中的词大多在正文中会多次出现。当然这种折中的方法有时也不够合理，它过分倚重于锚文本的关键词，常常被针对这种算法作弊的网页设计者利用。

对于方案 3 来说，一方面照顾到了方案 1 的索引压缩功能（文档按序存储）；另一方面照顾了重要的文档在检索过程中优先被检索的需要，而且防止了方案 2 中由网页设计者作弊可能带来的麻烦。有些网页设计者在网页中正文，锚文本中堆砌大量经常被查询的重要关键词，因此方案 2 在设计上不可避免的缺陷容易被利用。当然 PageRank 算法也会被作弊者利用，然而 PageRank 的作弊较为困难，所以方案 3 是较为理想的解决方案。

最后，总结一下正排索引和倒排索引的关系。本质上说，存在这样两个空间，一个称为“索引词空间”，一个称为“文档空间”。正排索引可以理解成一个定义在文档空间到索引词组空间的一个映射，任意一个文档对应唯一的一组索引词；而倒排索引可以理解成一个定义在索引词空间到文档组空间的一个映射。任意一个索引词对应唯一的一组该索引词其命中的文档。因此从文档到正排索引，进而从正排索引到倒排索引就是理顺这种关系的过程。使得给出一个索引词，就能通过倒排索引能够找到其命中的文档，以及位置信息。

4.5 数据规模的估计

在介绍索引系统主要概念后，通过索引数据规模估计的计算方法来体验索引系统设计中必须考虑的数据规模问题，本节首先从齐普夫法则（zipf law）开始说起。

4.5.1 齐普夫法则

齐普夫于 1902 年 1 月出生于一个德裔家庭（其祖父在 19 世纪中叶移居美国），1924 年以优异成绩毕业于美国哈佛学院。1925 年，齐普夫在德国波恩及柏林学习比较语文学专业。但是以其名字命名的定律却早已走出语言学，进入了信息学、计算机科学、经济学和社会学等众多研究领域，在学术界享有极高的声誉。

简单说来，齐普夫法则可以描述为第 k 个最经常出现的词，其词频（发生率）与 $1/k$ 成正比。

齐普夫是这样得到这个定律的，首先选用一个有关 James Joyce Ulysses（不妨理解为一本中等篇幅的书籍）的用词数据，统计不同词汇的出现次数，并按照出现次数排序。他发现了一个惊人的现象，即一个词汇在词频上的排位（rank）乘以其

出现的次数惊人地接近于一个常数 C，而这个常数在乘以一个常数 10，即书籍的实际总词数 260 430，如图 4-7 所示。

Ⅰ Rank (*r*)	Ⅱ Frequency (*f*)	Ⅲ Product of Ⅰ and Ⅱ (*r* × *f* = *C*)	Ⅳ Theoretical Length of Ulysses (*C* × 10)
10	2 653	26 530	265 300
20	1 311	26 220	262 200
30	926	27 780	277 800
40	717	28 680	286 800
50	556	26 500	278 000
100	265	26 500	265 000
200	133	26 600	266 000
300	84	25 200	252 000
400	62	24 800	248 000
500	50	25 000	250 000
1 000	26	26 000	260 000
2 000	12	24 000	240 000
3 000	8	24 000	240 000
4 000	6	24 000	240 000
5 000	5	25 000	250 000
10 000	2	20 000	200 000
20 000	1	20 000	200 000
29 899	1	29 899	298 990

(Zipf 1949 :24)

图 4-7　齐普夫的统计结果

词频排名第 10 的词汇，在书中出现了 2 653 次，排名和出现次数的乘积为 26 530。排名 20 的词汇，在书中出现了 1 311 次。排名和出现次数的乘积为 26 220，可以看出它们的乘积是十分接近的。如果将其看做相等，结果数乘以常数 C2 得到总词汇数，有以下公式：

$$occurence(T) * rank(T) = \mathrm{C1}$$

$$\mathrm{C1} \times \mathrm{C2} = N$$

其中 *occurrence*(*T*)表示词汇 T 的出现次数，*rank*(*T*)表示词汇在全 *s* 部出现的词汇中词频排序，C1，C2 为一个常数。通过化简处理得到 *s*：

$$\left(\frac{occurence(T)}{N}\right) = \left(\frac{1}{\mathrm{C2}}\right)\left(\frac{1}{rank(T)}\right)$$

$$f = \frac{occurence(T)}{N}$$

$$k = rank(T)$$

这样就得到本节开头提到 ss 的齐普夫定律。即第 *k* 个经常出现的词 *T*，其词频

f 与 $1/k$ 成正比，Zipf 在[zipf 1949]中提出这是由于语言上的省力原则起作用的结果。有兴趣的读者可以进一步阅读文献[姜望琪 2005]了解省力原则的社会学原理。

4.5.2 布尔检索模型下的索引规模估计

通过学习齐普夫法则，接下来利用这个法则粗略地估计布尔检索模型下的索引规模，从实践的角度体验一下索引的规模[Stanford IR]。

首先，由齐普夫法则，第 i 个最经常使用词汇的频率和 $1/i$ 成比率，因此第 i 个最经常用词汇的频率为 c/i。

接着，假定全部汉语词汇数目为 $S = 50$ 万，因此有 $\sum_{i=1}^{500000} c/i = 1$。这不难理解，在全世界所有中文书籍中取出 50 万个互不相同的词汇，并统计全部中文书籍的实际总词数。然后分别计算这 50 万的词汇各自出现的数目，用数目除以总词数即得到词频。这 50 万词汇的词频总和可以看做分母（总词数）不变，分子（各自出现的次数）相加。分子之和恰好为总词数，因此除的结果为 1，以下公式中 S 表示 50 万。

在数学上，称 $H_k = \sum_{i=1}^{k} 1/i$ 为“调和级数”，因此 $\sum_{i=1}^{S} c/i = 1$ 可以简化为：

$$c \times H_s = 1$$

调和级数的计算公式为：

$$H_s \approx \ln s = \ln 500000 \approx 13$$

这个常数 c 通过计算得到 c=1/13。

综上，一个词频排序为 i 的词汇，它的实际词频大约是 $1/(13i)$。这里的词频排序可以认为是海量数据中的排序，而不是局限于一本书。对于搜索引擎来说，一个词汇的词频就是它在全部网页中出现的实际次数除以全部网页实际出现的词汇数。例如一个词汇只在一个网页中出现，且出现了 K 次。所有网页为 N 个，平均每个网页长度为 L 个词。这样词频为 $K/(N \times L)$，文档频率为 $1/N$，注意文档频率和词频在计算上的区别。

布尔检索模型在下一节详细介绍，这里认为一个关键词只记录是否出现在文档中，而不记录其出现的具体位置。即在如图 4-6 所示的倒排索引中，不考虑 NHits

和 HitList 字段的信息。接下来对倒排索引长度的估计中，默认倒排索引不包含 Nits 和 HitList 字段。

在仅符合布尔检索的条件下，对倒排索引的长度进行粗略估计。假定每个文档的平均词汇数为 1 000，则第 i 个最经常使用的词汇在文档中期望出现的次数为 1 000*1/13i=76/i。

回顾一下文档频率的计算，如果一个词汇在文档中出现 1 次以上，则在计算文档频率时该文档将被计数。文档频率的计算不考虑一个词在文档中出现多少次，而只需要出现即可。例如在 4 个文档中，“搜索引擎”出现在 3 个文档中，则“搜索引擎”的文档频率为 3/4=0.75。第 1 个最经常使用的词汇在文档中出现了 76 次，从期望值的角度看，必定出现在所有的文档中；第 2 个最经常使用的词汇，在文档中平均出现 76/2=38 次，同理也几乎必然出现在所有的文档中。因此得出这样的结论，即在字典中词频排名 1~76 名的词汇几乎出现在所有的文档中；词频排名 77~152 的词汇几乎出现在一半的文档中。这不难理解，如果将海量文档两份合为一份计算的话（每个文档的词汇数为 2 000），用前面方法不难得到词频排名 77~152 的词汇也是几乎必然出现，粗略地认为出现在一半的文档中。这样我们得到下面的一个关系，如图 4-8 所示。

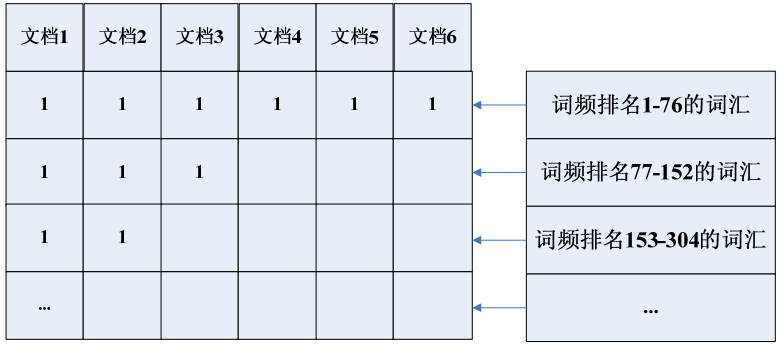


图 4-8　不同词频词汇的文档频率关系

在图 4-8 中用 6 个文档代表全部文档，标“1”表示词汇在文档中出现。可以看出词频排名 1~76 的词汇出现在全部文档中，词频排名 77~152 的词汇出现在 1/2 的文档中，词频排名 153~304 的词汇出现在 1/3 的文档中。

对于共计 500 000 个词汇，则有 500 000/76=6 500 这样的块。对于第 i 块，其出现在 n/i 个文档中。假定记录每个 DocId 需要 k 个字节，第 i 块（含 76 个词汇）需

要的空间为 76 *kn*/*i*，这样的块共有 6 500 个。不妨假设有 100 亿（10 GB）的网页，每个 DocId 的评价编码长度为 1 个字节（这里采用差分序列对 DocId 压缩编码，实际平均字节数略大于 1）。例如，在倒排索引中对词频排名为 1~76 的高频索引词，其倒排索引简化为如图 4-9 所示。

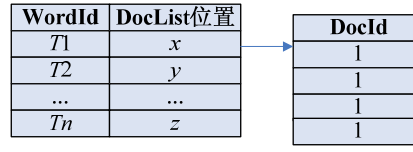


图 4-9　高频索引词在倒排索引中每个 DocId 的差值均为 1

在图 4-9 中由于高频索引词（例如 T1）几乎出现在所有的文档中，所以其指向的文档列表（doclist）几乎为 1，2，3，4，…，*n*，变为差分序列得到 1，1，1，1，…，1。由于序列间距足够小，因此对于大部分高频索引词，其所对应的每个 DocId 在压缩后所占的空间为 1 个字节。粗略地计算一下，采用 Variable Byte Coding 编码方式，小于 128 的数可用 1 个字节来表示。对于第 1 档高频词（共计 76 个），文档间距（gap）均为 1，因此需要 1 个字节存储文档间距；第 2 档的高频词（共计 76 个），文档间距为 2。由于小于 128，所以只需要 1 个字节存储文档间距。依次直到第 127 档的高频词，均采用 1 个字节来编码。这样共计 127×76=9 652 个高频词均可用 1 个字节表示。剩余的低频词出现在极少的文档中，因此可以认为平均每个文档间距的存储接近 1 个字节，这是一个重要的结论。

综上，在布尔检索模型下的倒排索引规模为：

$$\sum_{i=1}^{6500} 76\times 10\text{G}\times 1/i = 760\text{G}\times H_{6500} \approx 760\text{G}\times \ln(6500) \approx 6.7\text{TB}$$

一般情况下，我们可以得出这样的经验公式，即 1 MB 的文档大约需要 1 GB 的索引。为 100 亿（10 GB）网页创建倒排索引（不记录关键词出现的位置信息 HitList，只记录是否出现），大约需要 10 TB（1 TB=1024 GB）的索引空间。如果还需要存储位置信息（HitList），则需要的存储代价更加巨大。一般认为包含存储 DocId 和 HitList 信息后，索引的大小依然是 10 TB 这个数量级。如果按一台 100 GB 硬盘的服务器来计算，至少需要 100 台以上的服务器来做索引。而且为了一些安全稳定的因素，实际需要的服务器近千台。通过以上计算可以得出，索引主要面临的首要问题是“存得下”，下面我们将通过一些技巧来介绍如何存得下如此多的索引。

4.6　涉及存储规模的一些计算

前面提到对于索引目前 100 亿中文网页进行倒排索引需要 TB 级的存储空间，这么大规模的数据如何生成？在生成后如何存储？存储后如何支持高效的检索？这一连串的问题将在本节解答。

4.6.1　正排表与倒排表的合并

下载系统将抓取网页存放在网页库中，分析系统在分析后得到网页对象发送给索引系统，因此索引不停地得到这样的网页对象。由于网页对象在分析系统中进行了分词计算，最终分析的结果发往索引系统。

在索引系统中，通过计算不难得到如图 4-10 所示的正排表。在实际操作中，正排表不写入文件，而是保存在内存中。可以理解为内存中存储的正排索引，是正排索引的一种具体表现形式，同理，倒排表也是内存中存储的倒排索引，也是倒排索引的一种具体表现形式。

LId	WordId	NHits	HitList
Docx	T1	m	Hit1,...Hitm
	T4	n	Hit1,...Hitn
	NULL		

图 4-10　正排表

图 4-11 所示是一个倒排表，它们如何合并呢？

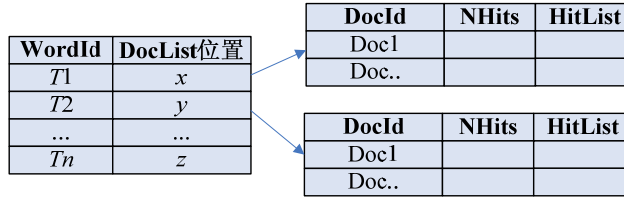


图 4-11　倒排表

正排表与倒排表合并的结果如图 4-12 所示。

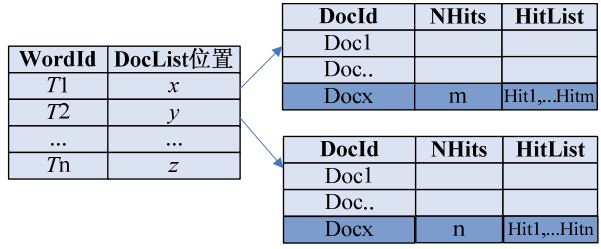


图 4-12　正排表与倒排表合并的结果

如图 4-12 所示，T1 是文档 Docx 的一个词汇。在左边的词典中找到 T1 的位置，然后通过其 Doclist 位置域存放的 DocList 位置信息找到记录表。并将文档编号（DocId）、在文档命中的次数（NHits），以及命中的位置列表（HitList）作为倒排表中的记录表中的一个记录。可以认为正排表和倒排表合并的过程，就是正排表中的数据追加倒排表的数据过程。追加后，正排表并不保留。而倒排表在内存中存储一定的记录后，成批顺序地写入磁盘，成为临时倒排文件（本章约定，在提到正排表时，表示其存放在内存中；而特指倒排文件时，表示其存放在磁盘中），如图 4-13 所示。

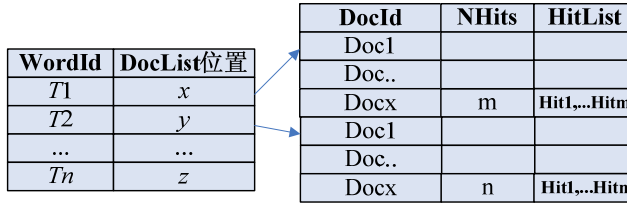


图 4-13　临时倒排文件

众所周知，由于磁盘的存储特性，所以很难在较大文件的中间追加数据。追加数据就不得不进行大量的数据移动，这种开销是极大的。回到这个例子中来，如图 4-13 所示的临时倒排文件中无法再追加有关 T1 的记录。

综上所述，生成倒排文件之前，倒排表（图 4-12）由于存放在内存中，因此可以任意追加数据。在顺序写入磁盘成为倒排文件后，倒排文件不再变化。但由于不断有新的数据需要进行索引，所以这样的临时倒排文件数量不可避免地还会不断增加。在导入全部需要作索引的数据后，索引系统会有多个这样的临时倒排文件，不妨假设共有 64 个这样的临时倒排文件。

出于性能上的考虑，在一次检索时只需要读取一个倒排文件即可得到全部与检

索词相关的索引信息。而不是依次读取全部 64 个临时倒排文件，然后进行结果组合。因此必须将这 64 个临时倒排文件归并成一个更大的倒排文件，称为“最终倒排文件”，其大小略小于 64 个临时倒排文件之和。因为在这 64 个临时倒排文件中分别保留了词典信息，所以这部分冗余的数据在归并成最终倒排文件后只需要保留一份词典即可。

在具体实现上，还有一种合并方法，简单说来就是将一个正排表（包含多个 DocId 的记录）通过对索引词排序的方法，然后一次性或者分批次地写入倒排表中。在第七节中会提到这个方法，我们这里先通过一个例子理解这个方法。首先正排表的结构需要进行调整，如图 4-14 所示。

LId	WordId	NHits	HitList
1	T1	1	1
1	T2	2	3,6
1	T3	1	7
2	T2	1	1
2	T3	1	3
3	T2	1	1

图 4-14　支持排序的正排表

这里的正排表和前面介绍的稍有不同，可以看到 LId 被冗余地存放。

在内存中保留一块这样的区域存放正排表，每增加一个需要索引的文档，就在正排表中追加一条记录。当追加足够多的记录后，正排表足够大，并符合批量做倒排表的条件后，按照关键词编号（WordId）进行一次稳定排序（稳定排序可以参考有关数据结构的书籍，例如归并排序就是一种稳定排序），得到如图 4-15 所示的排序后的正排表，这里认为 T1<T2<T3。

LId	WordId	NHits	HitList
1	T1	1	1
1	T2	2	3,6
2	T2	1	1
3	T2	1	1
1	T3	1	7
2	T3	1	3

图 4-15　排序后的正排表

在图 4-15 中，全部正排表记录按照 WordId 排序。由于采用稳定的排序，所以 LId 的顺序没有被破坏，图中 T2 关键词对应的 LId 依然是 1，2，3。由于在倒排表中 WordId 是顺序存放的，因此排序后的正排表可以依序逐个写入倒排表中，如图

4-16 所示。

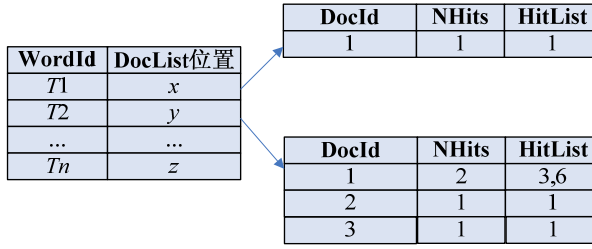


图 4-16　排序后的正排表逐个写入倒排表中

通过估计，每个临时倒排文件的大小大约为 100 MB，因此甚至可以不需要写入倒排表，而直接顺序写入磁盘中的临时倒排文件。这样就形成了内存中存放正排表，磁盘中放临时倒排文件的结果。

4.6.2　多个临时倒排文件的归并

首先考虑两个临时倒排文件的归并，通过前面数据规模估计，我们知道全部索引大小为 TB 量级的数据，后面将介绍倒排文件如何进行分布式存储。存储的节点数目控制在百这个数量级上，因此全部索引大小分布在 100 个索引结点上，平均每个索引结点大约需要存放 10 GB 的倒排文件（降两个数量级）。如果每个索引结点存放 64 个临时倒排文件，这样每个临时倒排文件约 100 MB（降两个数量级），64 个 100 MB 大小的临时倒排文件最终归并成一个 10 GB 大小的最终倒排文件是十分有挑战性的。每个临时倒排文件内部字典中词汇的编号是有序的，每个词汇对应的记录表（posting list）中的 DocId 是有序的，每个 HitList 中也是有序的，因此归并后依然要保持这种有序性。我们将这种归并过程笼统地称为“归并排序”，很显然这种规模的归并排序在内存中无法完成，下面介绍两种解决这个问题的方法。

（1）拉链法（Zipper）和二路归并。

（2）拉链法和多路归并。

无论是两路归并还是多路归并，拉链法都是必需的。我们首先考察拉链法和二路归并的组合，对于两个较大的文件进行归并排序，不可能将两个文件同时读入到内存中后进行排序。因此可以读取一部分归并，将结果以文件的形式写入磁盘，然后继续读取两个待归并文件的一部分。周而复始，直到读完其一种的某个文件，另

一个文件顺序写入结果文件。因此归并的方法如下。

（1）从头开始读取两个临时倒排文件的一小部分（例如每次读取 10 MB）。

（2）分别对 DocId 进行解压，将压缩后的差分序列还原成原始的差分序列。

（3）两个按照 DocId 有序的列表进行归并。

（4）归并的结果进行压缩。

（5）写入归并后的临时倒排文件 1&2 中。

以上方法如图 4-17 所示。

这里首先解压临时倒排文件 1 和临时倒排文件 2 的两块数据（只解压 DocId，不解压 HitList 的部分），分别解压后得到解压后的块 1 和解压后的块 2。因为 DocId 有序，因此归并可在线性时间内完成（参考有关数据结构的书籍中关于两个有序表归并的内容，这里不再展开）。归并后的倒排表中 DocId 依然有序，接下来还需要采用 Variable Byte Coding 编码方式对 DocId 进行压缩，最后写入结果倒排文件中（如图 4-17 的倒排文件 1&2）中。

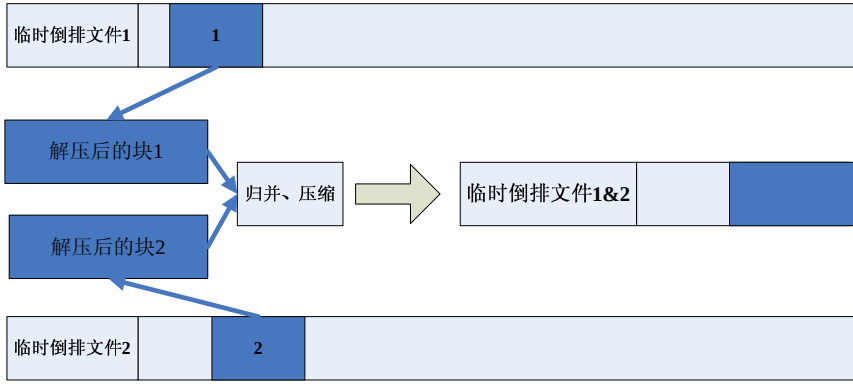


图 4-17　Zipper 方法

这种拉链的方法采用的思路是处理大文件（Big File）的一种通用思路，即每次仅取出大文件的一部分在内存中进行计算。计算结束后存储计算结果，并释放内存。继续读取文件的下一部分到内存、计算、存储计算结果并释放内存，周而复始直到处理全部数据。

64 个这样的临时倒排文件进行归并需要多少趟这样的两两归并呢？第 1 趟，64

个 100 MB 的临时倒排文件两两归并得到 32 个 200 MB 的临时倒排文件；第 2 趟，32 个 200 MB 的倒排文件两两归并，得到 16 个 400 MB 的临时倒排文件，直到最后剩下两个 3 200 MB 大小的临时倒排文件两两归并得到 1 个 6.4 GB 大小的最终倒排文件，这样整个归并的过程结束。每一趟归并结束后，归并段的个数少一半。

一般情况下，m 个初始归并段采用 k 路归并，则归并趟数为 $\lceil \log_k m \rceil$。如果 64 个初始归并段采用 2 路归并，则需要 6 趟（$\lceil \log_2 64 \rceil$）。由于每趟都不可避免的将全部文件换入内存和换出磁盘的过程，生成一个最终倒排文件的过程中需要产生大量的临时文件。大量的内存和磁盘数据的换入换出，所以整个索引制作的瓶颈主要在这里。通过下面这个例子来直观地认识归并的趟数对索引的巨大影响，如图 4-18 所示。

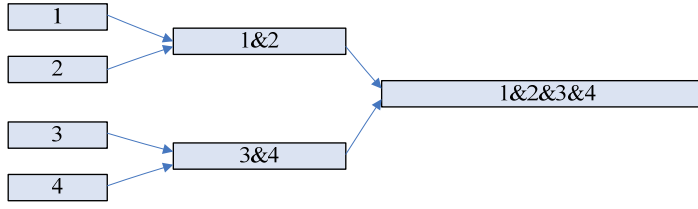


图 4-18　二路归并

图中假定有 4 个临时倒排文件，并且为 100 MB，在第 1 块和第 2 块进行归并时，需要读取的磁盘量为 200 MB，写入量也为 200 MB（块 1&2 的大小为 200 MB）。因此第 1 趟的归并（第 1 块与第 2 块归并，第 3 块与第 4 块归并）共计需要的磁盘 I/O 数量（读写量）为 800 MB。不难计算，第 2 趟归并的磁盘读写量也必然是 800 MB。实际上从宏观上看，每次都要读出全部的内容，计算。然后写入全部的内容，因此每趟的读写量是全部归并段的总数的两倍。这样 64 个 100 MB 大小的临时倒排文件进行 6 趟归并，需要的读写数量为 64×100 MB×2×6，约合 77 GB。

基于前面的分析，对于 m 个初始归并段，外排时采用 k 路归并，则归并趟数为 $\lceil \log_k m \rceil$。显然，随着归并路数 k 的增大，归并的趟数将减少。趟数的降低将大大减少磁盘 I/O 的数量，这样很自然地想到使用多路归并的方法。对于 64 个临时倒排文件的归并，如果能同时归并 64 路（k=64），则只需要一趟归并即可。

多路归并的方法在与数据结构相关书籍中都有介绍，通常采用“败者树”这样的数据结构，当然同时也需要采用拉链的思想。每次读取这 64 个临时倒排文件的一部分进行多路归并，直到处理全部的临时倒排文件，结束归并过程，这里不再展开

实现细节。

二路归并在实际处理中还有很多细节，例如每次解压出来的块中并不是全部都能够被归并写入结果倒排文件中会有一些块尾部分；例如解压后的块 1 的尾部包含了编号为 Tx 词汇及其对应的记录表（posting file），而在解压后的块 2 中没有包含编号为 Tx 这个词，那么这部分将需要等待下一次归并才能写入磁盘。问题是一次解压多少大小才使最合适，以及如何进行并发的归并操作等。

针对这些实现的细节，读者可以写一些模拟程序来体验大规模数据归并的全过程。另外可以参考[Managing Gigabytes]书籍官方主页中的一些及有价值的源代码。

4.6.3　倒排索引分布式存储

通过前面的学习，我们知道索引数据的规模为 TB 级。TB 相当于 1 000 GB，一个 1 000 GB 的文件是不可想象的。因此将全部索引文件存放在一台主机上，不仅是不合适的，而且是不安全的。这样一旦这个倒排文件损坏，全部服务就会受到很大影响，因此倒排索引的分布式存储技术应运而生了。

目前倒排索引分布式主要有两种方案，这里假定分布式的方法采用多索引结点（可以理解为多主机）的方法。即把一个巨大的倒排文件通过一些划分方法进行切分，使得每一个索引接点上只保留倒排文件的一部分。这样一方面加快了倒排文件的创建速度，降低了倒排文件损坏带来的损失；另一方面也提高了检索的效果。

多机分布式索引一般按照文档编号（DocId）或者按照索引词编号（WordId）进行划分。按照 DocId 划分的结果称为“局部倒排文件”（Local inverted file）；按照 WordId 划分的结果称为“全局倒排文件”（Global inverted file），如图 4-19 和图 4-20 所示。

图中每个索引结点可以理解为一台独立主机。由于索引被分布式地存储到不同的索引结点上，所以全局还是局部是相对于索引词来说的。对于局部倒排文件与索引词相关的一部分，DocId 被存放在一个索引结点的倒排文件中。换句话说，在图 4-19 中索引结点 A 中的倒排文件只存放了某个关键词的一部分匹配的 DocId；而全局倒排文件则存放了一个关键词全部匹配的 DocId。为了便于表述，以下称这两种方案分别为“局部方案”和“全局方案”。

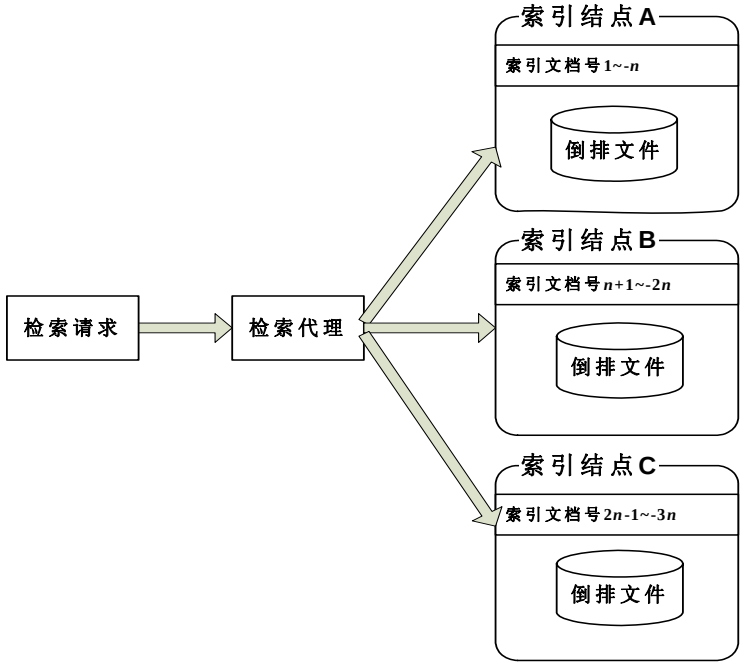


图 4-19　局部倒排文件

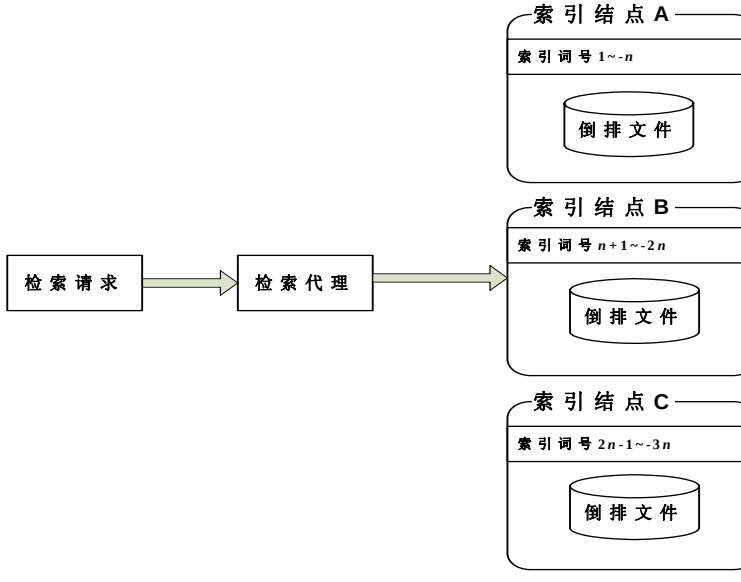


图 4-20　全局倒排文件

对于局部方案，索引结点按 DocId 的不同将倒排文件分布式地存储在不同的索引节点上。每个索引结点负责对一个 DocId 区间的文档进行索引，因此每个索引结点的倒排文件中的 DocId 互不相同。在检索时，将检索请求广播到每个结点。每个结点分别查询，最后由检索代理合并查询结果。

对于全局方案，索引结点按索引词 ID 的不同将倒排文件分布式地存储在不同的索引节点上。每个索引结点负责对一个索引词 ID 区间的文档进行索引。因此每个索引结点的倒排文件中的索引词 ID 互不相同，然而每个索引结点的文档可能重复。如果某一个文档的索引词足够多，以至于能够覆盖两个以上的索引词 ID 区间，则可能会被存放在多个索引结点上，这种重复存储将不可避免。

全局方案带来的好处是如果只检索一个单词，那么只需要在一个索引结点中检索即可。在图 4-20 中，对于一个检索请求，发现这个检索词在索引结点 B 中索引。因此整个检索只在索引结点 B 中完成，这就大大节约磁盘 I/O。此外，由于查询可能是不同的查询词，因而被分布在不同的索引结点上。这样并发的用户查询不需要在检索代理中排队，可以并发地查询从而提高效率。例如查询“XML”时，检索代理检测发现这个词存放在索引结点 A 上；查询“NJU”，检索代理发现这个词存放在索引结点 B 上，因此这两个关键词的查询可以做到并发查询。如果把索引看做某种公众服务，则全局方案是 64 个窗口同时对外服务，而局部方案是单窗口排队服务。

文献[Melnik et al. 2000]阐明了采用局部倒排文件的方案是相对有利的，主要是以下两点。

（1）可靠性高。

（2）降低网络负载，提高查询效率。

首先，对于全局方案，如果某个索引结点出现故障，可能导致某一些关键词无法查询；而局部方案在这种情况下（某个索引结点出现故障），最多是损失来自于一个 DocId 区间内的文档。对于查询效果的影响不大，因此该方案提供了足够的可靠性。

其次，对于全局方案，由于所有的查询结果来自于一个索引结点，因此检索代理要等待这个索引结点传输全部的查询结果，这是低效的。而局部方案中多个索引结点同时将查询结果并发传送，从而充分利用了网络带宽。有意思的是局部方案有利于并发地获取检索结果；而全局方案有利于查询并发，这一点请读者细细体会这种平衡的奥妙，因此局部方案在获取查询结果方面是有优势的。

到底哪一种方案最佳呢？在业界，一般认为局部方案的可靠性是必须的，因此主要应用了该方案；而在研究界，有研究表明[B2S Jeong et al 1995]，在多处理器多磁盘系统下，如果检索词均匀地被请求或者索引词分布偏差不大的情况下，由于避免了局部方案中检索请求必须排队的弊端，因此全局方案在性能上是最佳的。

4.6.4 倒排文件缓存

一般认为一个词被查询的频率与其被使用的频率相当，即频率高的词往往也是查询的热词，查询的频率依然符合齐普夫法则。即查询频率排名为 i 的关键词，其查询的实际频率与 1/i 成比例。大量的实验科学证明，在一段时间内那些有机会被检索到的检索词总是少数的，将这些少数的检索词存放在内存中可以大大降低读取磁盘中倒排文件的机会。关于倒排文件的缓存，可以参考文献[李晓明 2004]。这里只给出一个结论，如果一个索引结点需要 10 GB 的倒排文件，那么在这个 10 GB 的倒排文件中，只有不到 20%的索引词及其记录表应该进入缓存。然而这 20%的索引词占用的空间几乎是 80%，即需要 8 GB 的内存，这显然是难以实现的。因此业界采用了很多特有技术来完成这个工作，由于超出本书的范畴，所以不再深入下去。

倒排文件的缓存及第 5 章中提到的搜索结果页缓存的基本原理大致相当，读者可以参考第 5 章中的相关内容，深刻地理解在查询系统和索引系统这种缓存机制的重要性。

4.6.5 倒排索引词典统计信息的计算

倒排文件中的词典还需要有关每个索引词的统计信息，主要是词汇出现的文档数。这些信息主要用在查询系统中，在下一章中会详细介绍这些统计数据是如何应用的。

在索引系统中，这些关于索引词出现的文档数的统计是在查询请求发生之前预先计算好的，是倒排表的词典部分中不可分割的一部分。把做统计工作的这个模块称为“统计员”。关键词的文档频率的统计信息是全局的，因此在整个系统中仅有一个服务模块来完成这项工作。在系统结构图中，统计员的位置如图 4-21 所示。

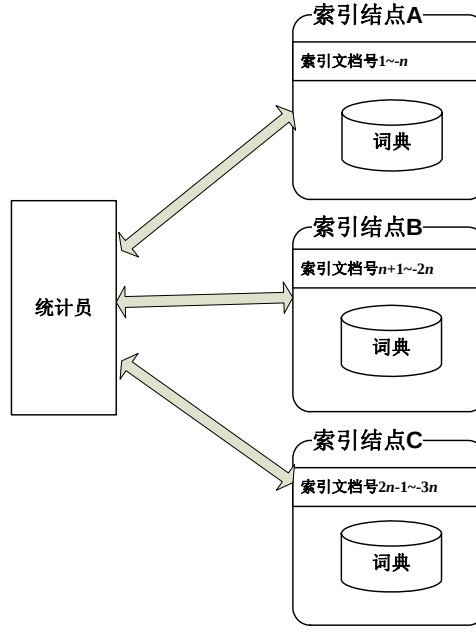


图 4-21　统计员的位置

如图所示，统计员把各个索引结点的词典信息综合起来做全局统计。然后将统计结果传回各自的词典中，继而保存这些全局统计信息。在下一节说明倒排索引创建过程时会详细提到关于统计信息的计算过程。

4.7　倒排索引文件的创建过程

倒排索引文件的创建过程更像是一个工程建设，其中大量应用了批量计算及流水计算的技巧，完成如此大规模的倒排索引文件的创建一直是搜索引擎的核心难点。

4.7.1　创建倒排表

首先通过一个例子完整地体会单个索引结点倒排文件的创建过程，如图 4-22 所示。

这里的网页库支持顺序访问模式（参见第 2 章第 2.5 节），能够顺序读取存放在其中的文档。为了简化，我们假定读取的文档按照顺序分别为文档 1、文档 2 和文

档 3，其正文内容分别为“rat dog”、“dog cat”和“rat dog”。通过在内存中完成正排得到索引词出现的文档和位置信息，例如，rat（1,1）表示在文档 1 的第 1 个位置出现“rat”这个索引词。接下来通过对字母排序（汉字可以按照汉字词汇编号排序），得到一个临时的按照索引词有序的结构，这有助于顺序写入各个索引词对应的记录表。在图 4-22 所示的倒排表达到一定大小（例如 100 MB）时，将倒排表顺序写入到临时倒排文件中。完成全部网页库的索引工作后，将产生的多个临时倒排文件归并为一个最终倒排文件（图 4-22 中忽略临时倒排文件归并的过程）。

从计算的角度上讲，临时倒排文件创建的全过程包含了磁盘读取（loading）、计算（processing）和写入磁盘（flushing）这 3 个过程，磁盘读取从网页库中读取一个个的文档；计算过程包括了正排计算、排序及归并等；写入磁盘主要是写入临时倒排文件。如果采用多道并行处理，则可大大提高索引创建的效率。

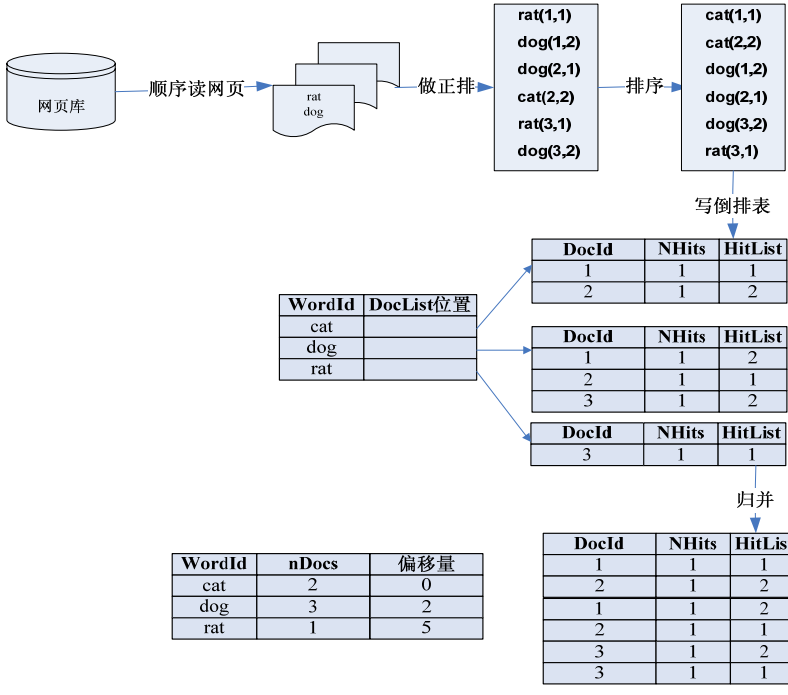


图 4-22　单个索引结点倒排文件的创建过程

如果将整个临时倒排文件创建过程抽象为 L（loading）、P（processing）和 F（flushing）3 个顺序的操作，即到写入临时倒排文件为止，则 L 操作和 F 操作是磁盘密集型操作；而 P 操作是 CPU 密集型操作。为了简化，采用两个线程并发操作的

处理方法，如图 4-23 所示。

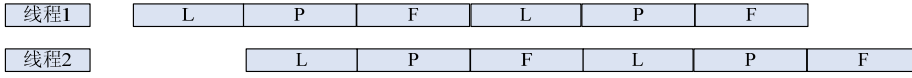


图 4-23　两个线程并发操作

这里线程 1 首先开始执行 L 操作，在执行 P 操作时磁盘空闲，这时线程 2 使用磁盘执行 L 操作。理论上最佳的效果是无论在什么时间点上，总是一个线程占用 CPU；另一个线程占用磁盘，这样相互配合可以高效地完成索引创建过程。这里还有一个问题就是一次 L 操作读取多少文档才是最佳的，可以使得这种合作能够最少出现互相等待的情况。对于两道线程并发，一个 L 操作读取的文档规模应该满足。使得 P 操作执行的时间与执行一次 L 操作的时间相当，即线程 1 的 P 操作能够和线程 2 的 L 操作重叠。当然实际处理中还有很多技巧，读者可以参阅文献[Arvind Arasu et al. 2001]获取更加详细的技术细节。

4.7.2　计算统计信息

计算统计信息在上一节中提到，这里给出两种计算方法，两种方法各有优劣。第 1 种方法从排序后的正排表开始统计；第 2 种方法从临时倒排文件统计。分别来看这两种方法的区别。首先通过图 4-24 来理解第 1 种方法。

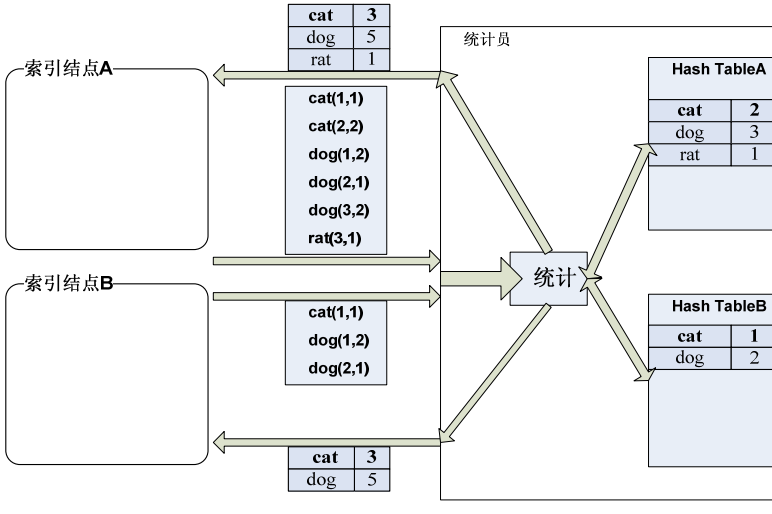


图 4-24　基于全局索引词出现文档数统计方法

内存中经过排序的正排结果在转换为倒排表之前，发给统计员一份复制。统计员为每个索引结点建立一个哈希表，这个哈希表用来进行计数。在全部网页库中的文档被处理完后，统计员将各个哈希表中的词进行综合统计，把相应的结果发给各个索引结点。注意这里发给索引结点 A 的统计结果和发给索引结点 B 的统计结果是不同的，因为索引结点 B 不包含“rat”这个索引词，因此没有必要把“rat”的信息发给它。这种方法由于需要维护哈希表的代价，因此需要耗费一定的内存空间，这是其主要缺点。

第 2 种统计方法主要采用基于已经计算好的倒排表数据来进行综合统计，整个过程相对简单。相当于对各个索引结点自身的统计结果进行综合统计，然后回传给各个索引结点。这种方法的主要缺点是需要等待最慢的索引结点做完索引后才能开始进行计算。

在完成了创建最终倒排文件和词典后，全部倒排索引文件创建工作完毕。从某种角度上看，这些都是一种预先计算（precomputation）。这种预先计算都是在为查询时节省时间，海量数据完成一次最终倒排索引文件的制作是非常耗时的，这些尽可能预先完成的计算为查询争取了宝贵的时间。

现在离搜索越来越近了，下一章我们来到搜索引擎最直接面对用户的查询系统，继续了解有关搜索及查询的知识。

参考文献

[Arvind Arasu et al. 2001] Arasu, A. and Cho, J. and Garcia-Molina, H. and Paepcke, A. and Raghavan, S. Searching the Web, ACM Transactions on Internet Technology，42 pages.

[B2S Jeong et al 1995] B2S Jeong ,E Omiecinski . Inverted file partitioning schemes in multiple disk systems . IEEE Transactions on Parallel and Distributed Systems.

[Managing Gigabytes] http://www.cs.mu.oz. au/mg/.

[Melnik et al. 2000] MELNIK, S., RAGHAVAN, S., YANG, B., AND GARCIA-MO LINA, H. 2001. Building a distributed full-text index for the web. In Proceedings of the Tenth International Conference on The World-Wide Web.

[S. Brin 1998] S. Brin and L. Page. The anatomy of a large-scale hyper textual Web

search engine. Computer Networks and ISDN Systems, 30（1-7）:107-117，1998.

[Stanford IR]http://www.stanford.edu/class/ cs276a/.

[Witten 1994] I. H. Witten, A. Moffat, and T. C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images. New York, NY: Van Nostrand Reinhold，1994.

[Williams 1999] H. E. Williams, J. Zobel. Compressing Integers for Fast File Access. The Computer Journal, 1999, 42（3）.

[Zipf 1949]Zipf G K. Human Behavior and the Principle of Least Effort . New York :Addion2Wesley，1949.

[李晓明 2004] 李晓明，闫宏飞，王继民，“搜索引擎——原理、技术与系统”，科学出版社.

[姜望琪 2005] 姜望琪“Zipf 与省力原则[J]”，同济大学学报：社科版，2005,16（1）:87-94.

第 5 章　搜索引擎的查询系统

5.1　知识准备

在搜索引擎 4 大系统中，第 4 个系统称为“查询系统”。查询系统直接面对用户，在接收用户的查询请求后，通过检索、排序及摘要提取等计算，将计算结果组织成搜索结果页返回给用户。整个查询过程不仅要快，而且必须能够提供用户满意的查询结果。文献[S. Brin 1998]提到“The goal of searching is to provide quality search results efficiently.”因此，本章从效率和效果这两个角度进行探讨，首先一起来了解一些关于查询系统的常用术语。

5.1.1　什么是信息熵

信息是个很抽象的概念，直到 1948 年，香农提出了“信息熵”（shāng）的概念才解决了信息的量化问题。

回忆一下数据结构中介绍过的哈夫曼编码，该编码通过计算不同词汇的词频并依据大小关系构建哈夫曼树，通过哈夫曼树为不同词频的词汇创建不同长度的前缀编码。从直观上看，高词频词汇编码较短，低词频词汇编码较长，这些编码都是 0 和 1 组成的比特串。直观的感觉还难以揭示事实的真相，下面通过一个例子来揭开编码长度和概率之间的联系。

假定红军和蓝军进行战术演习，红军打算左右夹攻蓝军，那么为了使得红军的兄弟部队能够相互通信，需要事先商量进攻的口令。并且由此确定是否同时发起攻击，取得更大的战果。假定有如下 3 类通信口令。

（1）如果太阳围绕地球转，就发动攻击。

（2）要么在白天攻击，要么在黑夜攻击。

（3）在 0~6 点或者 6~12 点或者 12~18 点或者 18~24 点攻击。

不考虑信息安全的情况下，红军的两支部队需要如何准备这些消息编码呢？显然编码越短，越有利于战场恶劣的条件，并节约通信成本。下面就对这 3 种情况分别进行分析。

对于第 1 种情况，红军两支部队不需要任何通信。因为太阳围绕地球转，这是肯定的。通信代价为 0，这样无论两支部队选择何种攻击方式都是合理的。

对于第 2 种情况，红军两支部队需要约定通信方式。假定为红 A 部通过通信网络传送一个比特 0，表示在白天发动攻击；传送一个比特 1，表示在黑夜发动攻击。显然，此时的通信代价为 1 个比特，1 个比特能够表达两种可能性。由于红军两支部队保证在白天或者黑夜攻击，而不会出现一支部队选择白天，一支部队选择黑夜进行攻击的可能性。因此这种信息更加有价值，然而也付出了 1 个比特的通信开销，不妨认为这个信息“值”一个比特。

对于第 3 种情况，虽然相对复杂，但也可以以如下 4 种编码方式约定口令。

（1）00：在 0~6 点攻击。

（2）01：在 6~12 点攻击。

（3）10：在 12~18 点攻击。

（4）11：在 18~24 点攻击。

此时的通信成本为两个比特，两个比特能够表达 4 种可能。这样红军两支部队的进攻同步性更强，其先后进攻的时间最多差 6 小时，很显然这种信息比第 2 种更加有价值。同样道理，认为这个信息“值”两个比特。

综合上述 3 种情况的分析，从直观上看，信息包含的情况越多，信息越有价值，需要的通信代价就越大。信息是否有价值隐约地和概率有着密不可分的关系，那么如何衡量发起攻击这个信息，如何从直观跨越到客观呢？

1948 年，香农长达数十页的论文“通信的数学理论”成为信息论正式诞生的里程碑。在其通信数学模型中清楚地提出信息的度量问题，得到了如下著名的计算信息熵（Entropy）公式：

$$H(X)=\sum_{i=1}^{n}-p_i\log(p_i)$$

对于上例中的第 3 种情况，假定每一种攻击可能均等，均为四分之一，那么何时发动攻击这样一个随机变量 X 包含 4 个随机事件，概率分别表示为：

$$P(X=\text{“在 0~6 点攻击”})=\frac{1}{4}$$

$P(X=$“在 6~12 点攻击”$)=\frac{1}{4}$

$P(X=$“在 12~18 点攻击”$)=\frac{1}{4}$

$P(X=$“在 18~24 点攻击”$)=\frac{1}{4}$

那么

$$H(X)=-\frac{1}{4}\log_2(\frac{1}{4})+-\frac{1}{4}\log_2(\frac{1}{4})+-\frac{1}{4}\log_2(\frac{1}{4})+-\frac{1}{4}\log_2(\frac{1}{4})=2$$

与前面计算的结果 2 个比特一致。

信息熵 $H(X)$在信息论中称为消息 X 的“熵”，其含义是信息集 X 发出任意一个随机事件的平均信息量。“熵”值 $H(X)$说明了消息集 X 的每个事件的平均存储的位数，即用多少个二进制表示一个消息。在约定进攻口令的这个例子中，4 个口令中发生一个口令平均需要的通信或者存储代价（平均信息量）为两个比特。

香农通过“熵”阐明了概率与信息的关系，即变量的不确定性越大，熵也就越大，将其搞清楚所需要的信息量也就越大。信息熵是一个十分重要的概念，下面在介绍经典的 TF/IDF 方法时还将就此问题继续展开。

5.1.2　检索和查询的区别

本章约定对于查询来说，适用于真实用户进行的一次查询是相对于搜索引擎查询系统而言的；对于检索来说，适用于检索代理对索引库进行的一次检索是相对于搜索引擎索引系统而言的。查询的结果是搜索结果网页，检索的结果是与查询词相关的文档列表（doclist）。

5.1.3　检索词和查询词的区别

严格意义上，普通用户提交给查询系的关键词称为“查询词”；经过查询系统分词，提交检索代理的称为“检索词”。例如用户提交查询词为“清华大学图书馆”，

通过分词，提交给检索代理变成“清华大学”和“图书馆”两个检索词。为了简化，本章并不区分查询词和检索词，而统一使用查询词这个术语。

5.1.4 自动文本摘要（Automatic Text Summarization）

自动文本摘要简称“自动摘要”，它是从文档中自动提取出的一个正文片断。用户仅仅需要浏览整个正文片段就能够了解文档中与查询词相关的部分，进而判断是否值得详细阅读整篇文档。

5.2 网页信息检索

网页信息检索的数据源来自于网页索引库（在前一章中介绍了网页对象被索引入库的全过程），网页信息检索输出是一组文档编号，这些被编号的文档都是索引库中包含查询词的文档。

5.2.1 早期的检索模型

早期的检索模型是一种称为“布尔模型”（Boolean Models）的检索模型。布尔模型也称为“集合模型”，是一种采用 AND、OR 及 NOT 等逻辑运算符将多个查询词连成一个逻辑表达式，继而通过布尔运算进行检索的简单匹配模型。例如查询词为“走进搜索引擎 检索模型 — 搜索”，将会被翻译成“走进搜索引擎 AND 检索模型 NOT 搜索”这样的逻辑语言。按照自然语言的翻译，这个逻辑语言表示包含“走进搜索引擎”且包含“检索模型”，却不包含“搜索”的文档集合。对于查询系统来说，这样的查询词表示用户请求检索包含“走进搜索引擎”且包含“检索模型”，却不包含“搜索”的文档集合。

布尔模型的这种检索易于实现，检索速度快。但是由于没有考虑文档和查询词的相关性问题，没有区分查询词的权重问题。因此在“效率”和“效果”的两难选择上放弃了“效果”，而仅仅考虑了“效率”。

此外，如果查询词中有一个关键词没有包含，则可能出现漏检。不妨通过一个例子来说明布尔模型的这些主要的缺点。

假定有如下这样 3 篇待检索的文档。

（1）在传统搜索引擎架构中，搜索引擎由 4 个系统构成，分别是下载系统、分析系统、索引系统及查询系统。

（2）机械行业内一般把小型挖掘简称为“小挖”，小挖由 5 个系统构成，分别是……，详细地理解这些名词可以使用 Google 搜索引擎搜索一下。

（3）搜索引擎有 4 个主要功能模块，分别是下载系统，分析系统，索引系统和查询系统。这 4 个系统是搜索引擎的核心，其中查询系统是搜索引擎唯一直接面对客户的系统。

如果采用布尔检索模型，在查询“搜索引擎 系统构成”这样的查询词时，文档 1 和文档 2 均会被检索到，因为文档 1 和文档 2 均包含了全部查询词。显而易见，在第 1 个文档中，“搜索引擎”这个关键词出现了两次，“系统构成”出现了 1 次；在文档 2 中“搜索引擎”出现了 1 次，“系统构成”也出现了 1 次，直觉上看应该是文档 1 的相关性更好。在布尔模型中很难进行相关性强弱的度量，它只解决“有”还是“没有”的问题，不解决“好”还是“不好”的问题。

最后，从用户查询意图上看，文档 3 比文档 2 更加符合用户的查询意图。文档 3 中出现了 3 次“搜索引擎”这个关键词，仅仅因为没有包含“系统构成”这个关键词，而没有被检索出，而文档 2 只是沾边提到了搜索引擎，却能够被检索出。

布尔模型归纳起来存在如下两大优点。

（1）表达简单且易于实现。在关键词检索的过程中，把检索计算转变为集合运算，特别是集合间的求交集运算和集合间的差运算。

（2）检索速度快。布尔模型的计算主要是集合求交运算，这将在下一节中介绍。

正是由于布尔模型的两个优点造成了布尔模型的如下两大不足。

（1）如果有一个查询词没有被包含，则检索失败。由于布尔模型表达简单，缺乏灵活性，造成上例文档 3 中没有包含“系统构成”这一关键词，因而无法被检索出来的情况。

（2）检索出来的结果很难进行相关性排序。由于布尔模型计算简单，例如前面的例子中检索“搜索引擎 系统构成”的过程中，文档 1 和文档 2 与查询词的相关性没有被计算，从而无法了解哪个文档更加符合用户的查询意图（通常认为符合用户

查询意图的文档在搜索结果中应排名靠前）。

布尔模型的不足主要由于没有考虑到关键词在查询中的权重问题，这一点不足在向量空间模型（Vector Space Models）中得到部分解决。虽然不够完美，但是已经足够解决上面例子中的检索问题。

5.2.2 向量空间模型（Vector Space Models）

和布尔模型不同，向量空间模型主要关心的是“效果”，而非“效率”。向量空间模型提出了将查询词和文档按照关键词的维度分别向量化，然后通过计算这两个向量间夹角余弦的方法得到文档与查询词的相似度。从而优先检索那些和查询词相似度大的文档，并且能够对检索出的文档按照与查询词的相似度进行排序。向量空间检索模型的计算方法如图 5-1 所示。

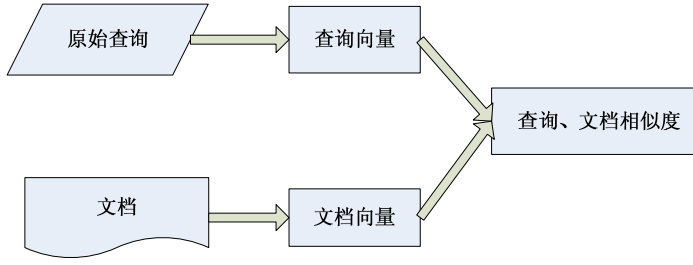


图 5-1 向量空间检索模型的计算方法

在向量空间检索模型中，通过下面 3 个步骤进行检索。

（1）把原始查询和文档都看做是文本，使用同样的向量化过程分别得到查询向量和文档向量。

（2）通过计算向量相似度的方法计算原始查询和文档的相似度。

（3）按照与查询词的相似度从大到小排序文档，返回给用户。

向量（vector）是一个很抽象的概念，它又称为“矢量”。最初被应用于物理学，很多物理量，如力、速度、位移、电场强度，以及磁感应强度等都是向量。大约在公元前 350 年，古希腊著名学者亚里士多德就已提出力可以表示成向量，两个力的组合作用可用著名的平行四边形法则来得到。英国大科学家牛顿最先使用有向线段表示向量。

事实上，向量包含了两层含义，即长度和方向。长度用向量的模表示，向量的模（长度）的计算公式为向量的每个分量的平方和开根号。由于向量具有方向，所以方向上的差异（角度）被用来量化向量的相似程度。

将各种不同的关键词看做是不同的维度，那么每个文档按照关键词进行向量化，得到向量中每一个分量可以理解为向量在各个关键词维度上的投影。这一点不难理解，三维坐标上描述一个点采用的方式为（*a*，*b*，*c*）表示向量在 *X* 轴上的投影为 *a*，在 *Y* 轴上的投影为 *b*，在 *Z* 轴上的投影为 *c*。在这里只是把代表三维空间中 3 个轴转换为 *n* 个关键词的 *n* 维空间，这样每一个查询句子和每一个文档都可以用这个 *n* 维空间来表示。

通过下面的一个例子来理解向量化的过程。假定汉语的词汇表只有“走进”、“搜索引擎”和“学习”这 3 个词（实际上，常用的汉语词汇过万），那么这 3 个词组成的向量空间就是我们熟悉的三维空间，如图 5-2 所示。

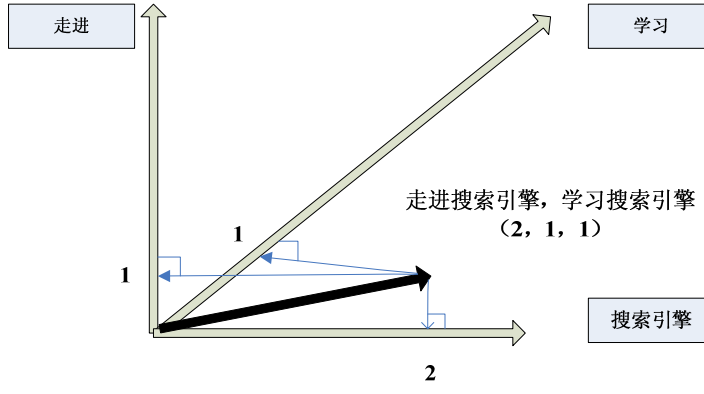


图 5-2　三维空间

在图中，对于“走进搜索引擎，学习搜索引擎”这个句子，通过计算每个词汇的出现的次数，得到这样的统计信息。即“搜索引擎”出现两次，“走进”出现 1 次，“学习”出现 1 次。将 3 个词的维度理解为三维空间的 *XYZ* 轴，这样“走进搜索引擎，学习搜索引擎”在词汇表构成的向量空间内表示为向量（2，1，1）。这个向量的 3 个分量的意义可以理解为对 3 个轴的投影分别是 2，1，1，物理含义为这些关键词在查询句子中分别出现的次数，同时注意这里向量的方向性用箭头表示。

现在我们扩展到四维空间上理解，假定词汇表中还包括了“检索模型”一词，这样对于“走进搜索引擎，学习搜索引擎”这个句子进行向量化的结果可能是（2，

1，1，0），其中四维空间的第四维表示“检索模型”。由于这个句子中没有出现“检索模型”，因此它在这个关键词维度上的投影为 0。

据统计常用汉语词汇大约 5 000 条，如果用这 5 000 维的词向量空间表示这个句子，可能是这样的形式，即（0，0，…0，2，0，…1，…1，…）。其中标“0”的分量表示句子在这个词汇上的投影是 0，或者说句子中没有出现这个关键词。由于句子中只出现 3 个词，因此在向量中只有 3 个分量为有效的非 0 值。可见在实际的计算中，向量通常都是十分稀疏的。

向量化的过程就是对一个文档按照关键词的维度进行向量化，每个向量的分量可以理解为包含这个词的权重（weight）。出现次数多的词权重就较大，对向量方向的影响力也较大。为了使不同文档和查询词的相关性具有比较性（相关性排序的需要），保证对大文档和小文档做到公平，还需要对关键词的出现次数做归一化的工作，即转化为词频（词数/总词数）作为向量的分量。因此在上面例子中，“走进搜索引擎，学习搜索引擎”中的“走进”的词频为 1/4，“搜索引擎”的词频为 2/4，“学习”的词频为 1/4。因此在如上图所示的关键词向量空间下，这个查询词被向量化为（2/4，1/4，1/4）。

事实上，向量中的每个分量同除以相同的数不会改变向量的方向，但是会改变向量的距离。因此在只考虑向量方向，而不考虑向量长度的情况下，没有必要使用词频作为向量的分量，这样反而引入了浮点计算的麻烦。考虑到其他可能需要进行向量距离运算的场合，以及为下一小节中的 TF/IDF 的权重量化计算做准备，提前了解词频的有关概念，并使用关键词词频作为向量的分量表示。

在向量化的工作完成后（下一节将提到实际上采用经典的 TF/IDF 方法进行向量化的工作），就需要解决计算文档和查询词相似度的问题。向量空间模型中一般采用向量之间的夹角余弦值作为向量是否相似的度量依据。

向量间的夹角余弦的计算公式：

$$\cos\theta = \frac{\boldsymbol{a} \bullet \boldsymbol{b}}{|\boldsymbol{a}| \times |\boldsymbol{b}|}$$

其中 $\boldsymbol{a}$,$\boldsymbol{b}$ 表示向量，$\bullet$ 表示向量的点乘，$|\boldsymbol{a}|$ 表示向量的模，或者说是向量的长度。

通过一个具体的例子理解这个计算过程。

假定在一个 7 个关键词的向量空间下，一个查询词向量化为 $\boldsymbol{a}$(0，0，2，0，1，0，1)，一个文档向量化为 $\boldsymbol{b}$(0，1，3，5，2，4，0)，夹角余弦计算方法如下：

$\boldsymbol{a} \bullet \boldsymbol{b}$ =(0，0，2，0，1，0，1)*(0，1，3，5，2，4，0)^T

= 0*0 + 0*1 + 2*3 + 0*5 +1*2 + 0*4 + 1*0

= 8

$$|\boldsymbol{a}| = \sqrt{0^2 + 0^2 + 2^2 + 0^2 + 1^2 + 0^2 + 1^2} = \sqrt{6} = 2.45$$

$$|\boldsymbol{b}| = \sqrt{0^2 + 1^2 + 3^2 + 5^2 + 2^2 + 4^2 + 0^2} = \sqrt{55} = 7.42$$

$$\cos\theta = \frac{\boldsymbol{a} \bullet \boldsymbol{b}}{|\boldsymbol{a}| \times |\boldsymbol{b}|} = \frac{8}{2.45 * 7.42} = 0.44$$

这样查询词 a 和文档 b 的相关性就转化为 0.44 这个具体的数值上，使得相似性成为可以量化的概念，因此相似性量化的结果称为“相似度”。

在实际计算中，如果向量 $\boldsymbol{a}$ 表示查询向量，向量 $\boldsymbol{b}$ 表示文档向量，在计算查询向量和一组文档向量的相似度时，查询向量总是不变的，或者说对每个文档向量来说查询向量都是相同的。因此相似度计算中是否除以 $|\boldsymbol{a}|$，对将来进行的相似度排序没有影响，可以作为公共因子消去。计算相似度实际只需要计算 $|\boldsymbol{a}| \times \cos\theta$ 即可，方法如下：

$$|\boldsymbol{a}| \times \cos\theta = \frac{\boldsymbol{a} \bullet \boldsymbol{b}}{|\boldsymbol{b}|}$$

其中每个文档向量的模可以预先计算并保存，而不需要每次查询都执行一次文档向量的模运算。这样，每次求相似度只需要一次向量点乘和除法计算即可。

对于两个高维稀疏向量（由于汉语词汇众多，实际向量化后的向量维数高，非 0 值少），向量的表示和向量点乘的计算也是需要一定技巧。可以采用哈希表的方法快速找到两个向量相同分量的非 0 值进行计算，这里不再详细展开。

为了简化描述，在此前提到的关键词量化过程中采用词频作为向量化中每个向量的分量，而事实上却采用了经典的 TF/IDF 方法为每个关键词进行更加合理的量化。下面我们将走进经典的 TF/IDF 方法，领略信息检索的精髓。

5.2.3 关键词权重的量化方法 TF/IDF

在实际的查询词及文档向量化应用中仅仅使用词频作为分量是不够的，例如我们曾经使用的例子中，查询“搜索引擎 系统结构”无法检索出文档 3。这是因为没有突出“搜索引擎”的重要性，所以还是无法取得良好的检索效果，接下来将介绍搜索引擎中经典的 TF/IDF 权重计算方法。

首先，我们来继续本章开头提到的信息熵这个概念深入理解概率与信息的关系。下面我们科学地给出“自信息”和“熵”的概念。

定义 1（自信息）：任意随机事件的自信息量定义为该事件发生概率的对数的负值，设该事件 x 的概率为 $p(x)$，那么其自信息定义为：

$$I(x) = -\log(p(x)) = \log(\frac{1}{p(x)})$$

自信息也可以理解为某个概率的事件进行编码需要的最小编码长度。

定义 2（熵）：在信息论中自信息量是一个随机变量，它不能用来作为整个信源的信息测度。因此我们引入平均自信息量，即熵，定义为：

$$H(X) = \sum_{i=1}^{n} -p_i \log(p_i)$$

Claude Shannon（香农）的源代码[Shannon,1948]理论指出，最理想的编码方法是词汇表中第 i 个词汇预期出现的概率为 p_i，那么该词汇需要分配 $-\log_2(p_i)$ 个比特长度的编码。最佳编码符号中的比特数目表示符号的信息内容（information content），整个词汇表中的全部词汇的信息量的平均大小称为“概率分布的熵”，即：

$$E = \sum_{i=1}^{n} -p_i(\log_2 p_i)$$

E 用比特/符号为单位表示，表示词汇表平均每个词汇需要的加权平均编码长度[Baeza-Yates et al,1999]。

在编码中用“熵”值衡量是否最佳编码。若以 W 表示采用一种编码方式后词汇表平均编码长度，则可能情况如下。

（1）$W>H(X)$：有冗余，不是最佳编码。

（2）$W<H(X)$：不可能。

（3）$W==H(X)$：最佳编码（一般 W 稍大于 $H(X)$）。

这里 $\sum_{i=1}^{n} p_i \times l_i$　p_i 为词汇表中的第 i 个词汇的发生概率（文档频率），l_i 为给字母表中的第 i 个词汇的实际编码位数，因此“熵”值是平均编码长度的下限值。数据结构中提到的哈夫曼编码就是对“熵”值的极限逼近，由于实际编码长度不能为小数，因此哈夫曼编码所能达到的压缩比距离压缩极限还有一定距离。

熵最大限度地压缩冗余的信息对于衡量关键词权重具有特殊意义，我们通过一个例子循序渐进地来理解为什么要把熵这个概念引入到关键词权重的量化计算中来。

为了简化，假定汉语的词汇表中总共仅有 3 个词汇，分别为“走进”、“搜索引擎”及“学习”，对这 3 个词汇在实际应用中使用的情况进行概率统计，并且分别计算其各自出现的概率为 1/256、1/4 096 及 1/256。因此按照熵的计算公式，汉语词汇表中的词汇平均编码长度为：

$$\frac{1}{256}\log_2(256)+\frac{1}{4096}\log_2(4096)+\frac{1}{256}\log_2(256)=0.0654。$$

接下来，假定一个文档就是一个信息源。这个文档包含了 T_1，T_2，T_3，…，T_n 共 n 个词汇。每个词汇各出现了 N_1，N_2，N_3，…，N_n 次，其中在海量文档中出现的文档频率（词汇的发生概率）分别为 D_1，D_2，D_3，…，D_n，那么传输一个这样的文档需要多少编码长度呢？

假定每个词汇的出现相互独立，并且不考虑出现的先后顺序，因此由这些词汇组成这篇文档的概率为：$X=D_1^{N_1}\times D_2^{N_2}\times\ldots\times D_n^{Nn}$（概率的乘法公式）。参考自信息的公式，对具有这种概率的事件进行编码，需要的编码长度为：

$$-\log(X)=-\log(D_1^{N_1}\times D_2^{N_2}\times\ldots D_n^{N_n})$$

$$=-N_1\times\log D_1-N_2\times\log D_2-\ldots-N_n\times\log D_n$$

因此这篇文档理论上能够最大极限地被压缩到−log(X)个比特。数学上可以证

明，这样的压缩形式是极限压缩。所有的压缩算法都参考对熵压缩的逼近程度来量化压缩能力的大小，在数据结构中学过的哈夫曼编码就是对信息熵的一种逼近编码。另外上面的公式也可以这样理解，对于关键词 T_i，其文档频率为 D_i。这个关键词的编码长度为 $-\log D_i$，在文档中该词出现了 N_i 次，因此总共需要的编码长度为 $-N_i \times \log D_i$。而全部文档需要的编码长度为 $\sum_{i=1}^{n} -N_i \times \log D_i$，这与前面的计算结果相同。接下来我们考察该文档中每个词的平均编码长度。

很明显，平均编码长度为全部文档需要的编码长度除以总词数，即：

$$\frac{\log(X)}{\sum_i^n N_i}$$

展开这个计算式得到：

$$\frac{-N_1 \times \log D_1 - N_2 \times \log D_2 - \ldots - N_n \times \log D_n}{N_1 + N_2 + \ldots + N_n}$$

在这个平均编码长度中，各个关键词都做出了不同的贡献。我们将关键词在文章中的重要性量化为对平均编码长度的贡献上，不难得出这样的结论，即越是出现次数多（词频高）且罕见的词汇（文档频率低）对平均编码长度大小的贡献越大。

假定一个文档总词数为 $K(K=\sum_i^n N_i)$，对于关键词 T_i 来说，它对平均编码长度做出的贡献为：

$$\frac{-N_i \times \log D_i}{K}$$

进一步转化这个表达式得到：

$$\frac{N_i}{K} \times \log \frac{1}{D_i}$$

其中：$\frac{N_i}{K}$ 为在文档中关键词 T_i 的词频（TF，term frequency），$\log \frac{1}{D_i}$ 为文档中关键词 T_i 的文档频率倒数的对数式，称为“文档频率”（IDF，inverse document frequency）。上面这个平均编码长度的公式可以改写为：

$$\frac{-N_1 \times \log D_1}{K} + \frac{-N_2 \times \log D_2}{K} + \ldots + \frac{-N_n \times \log D_n}{K}$$

显然不同关键词的 TF×IDF 值对编码长度的最终形成做出了不同的贡献，越是出现次数多（词频高）且罕见的词汇（文档频率低）对平均最终计算得到的平均编码长度的大小贡献大，这就是经典的 TF/IDF 方法。

文档频率是从哪里来的呢？回顾前面搜索引擎索引层的倒排索引词典统计信息中提到的文档频率的计算方法，在索引阶段这些数据已经计算完毕，因此这里可以直接使用文档频率来进行权重计算。对于“走进搜索引擎，学习搜索引擎”这个句子，假定“走进”的文档频率是 1/256，“搜索引擎”的文档频率为 1/4 096，“学习”的文档频率为 1/256。那么它们各自的权重分别如下。

（1）搜索引擎：$\frac{2}{4} \times \log(4\ 096) = 6$。

（2）走进：$\frac{1}{4} \times \log(256) = 2$。

（3）学习：$\frac{1}{4} \times \log(256) = 2$。

这样一个句子在仅由这 3 个词构成的向量空间中被量化为（6，2，2），和前面仅使用词频作为量化标准的（2，1，1）相比，突出了“搜索引擎”这个关键词在句子中的重要性。使得特别是一些诸如“我们”及“他们”这样的高频词能够很好地降低权重，而提高了低频词的权重。并且使得具有不同权重的词汇在进行向量相似度计算过程中发挥不同的作用，使得量化的结果更加科学。

5.2.4　搜索引擎采用的检索模型

搜索引擎采用了布尔模型和向量空间模型结合的方法来进行信息检索，布尔模型的检索效率高且易于实现；向量空间模型能够提高检索的相似度，通过相似度排序的手段能够大大改善查询效果。因此搜索引擎将两者的优势相结合，完整的检索过程如图 5-3 所示。

图中方块为计算部分，斜方块为数据部分，详细的检索过程如下。

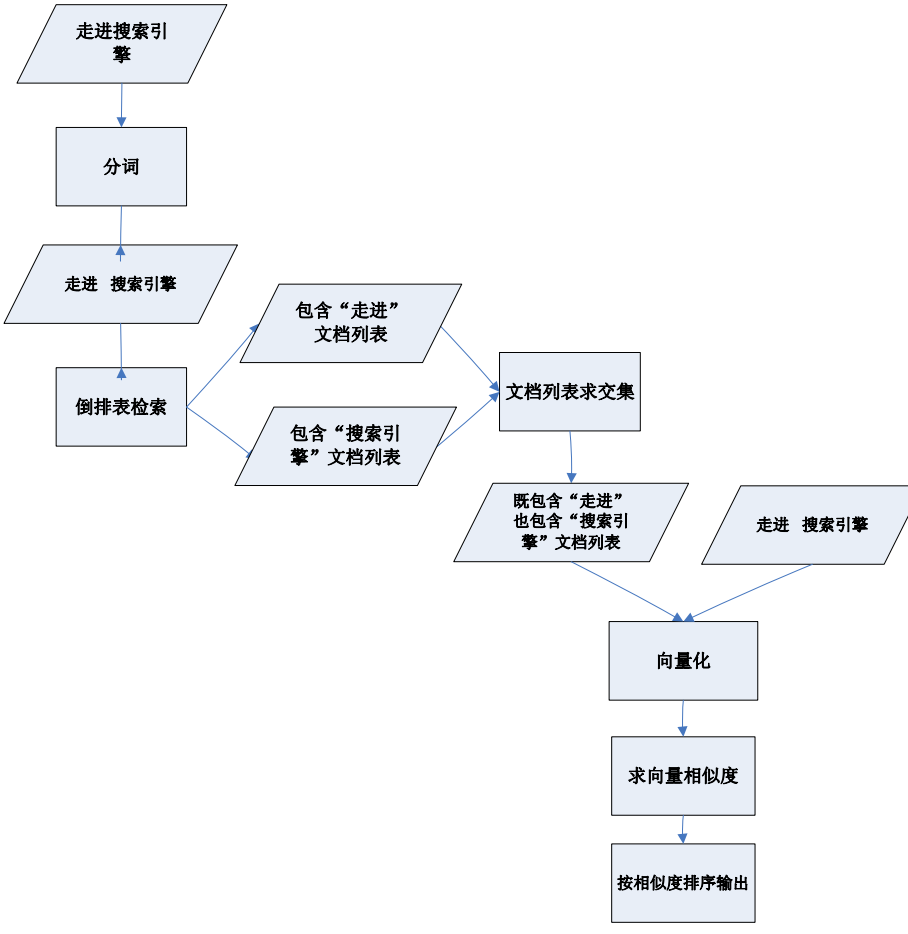


图 5-3　完整的检索过程

（1）对查询词进行分词，得到一个逻辑表达式。例如查询“走进搜索引擎”，将会被切分成“走进”，“搜索引擎”这两个词。并且转换为用 AND 逻辑表示的表达式，即“走进”AND“搜索引擎”。

（2）采用布尔模型的方法得到结果文档列表，例如从倒排索引中提取包含“走进”关键词的文档列表和包含“搜索引擎”关键词的文档列表。并将检索出的文档列表求交集，得到既包含“走进”，也包含“搜索引擎”的文档列表。

（3）将步骤（2）得到的文档列表中的全部文档和查询词分别向量化，并求向量间的相似度。

（4）按照相似度排序输出检索结果。

综上所述，全部过程包括分词、doclist 求交、向量化并求向量夹角及排序这 4 种计算，并且这 4 项计算依次完成。

在查询系统我们介绍过分词的计算，这里不再重复，接下来的两个小节将依次介绍文档列表求交的计算方法和结果排序的技巧。

5.2.5　多文档列表求交计算

在实际的查询中，包含一个或者多个查询词。有时一个查询词也会因为分词而分解出多个词，因此可能包含如下 3 种情况。

（1）查询单个词：例如查询“中国”。

（2）查询多个词：例如查询“中国 搜索引擎”，搜索引擎默认查询词中间空格表示用户主动的分词，认为是一次多词查询。

（3）查询一个词：由于被分词，而成为实际的多词查询。例如查询“走进搜索引擎”将会被分解成为“走进 搜索引擎”，本质上和第 2 种类型相当。

对于第 1 种情况，只需要在倒排索引表中检索出一个关键词对应的文档列表。由于检索结果是单个文档列表，因此不需要进行多文档列表求交的计算过程。

对于第 2 种和第 3 种情况，都需要进行多文档列表的求交计算。例如在倒排索引表中检索出包含“走进”一词的文档列表为 doclist_1(1，5，9，12)，表示这 4 个文档编号的文档含有“走进”这个词汇。同理假定包含“搜索引擎”的文档列表为 doclist_2 (5，7，9，11)，这样同时包含“走进”和“搜索引擎”这两个关键词的文档为 $doclist_1 \cap doclist_2$=(5，9)。“∩”表示文档列表间的求交运算符。

让我们从具体到抽象，这种在布尔模型中多个查询词而导致的多文档列表求交运算形式化的描述为由一次查询 $Query < T_1, T_2, ..., T_n >$ 引起的查询。首先在倒排索引表检索出文档列表组 $\text{Doclists} < doclist_1, doclist_2, ..., doclist_n >$，其中从 T_i 检索出的文档列表为 doclist_i。由这次查询返回的结果为 $doclist_1 \cap doclist_2 \cap \ldots \cap doclist_n$，查询返回的结果文档中包含了全部查询词。

这里介绍一种多文档列表求交的方法，这种方法称为“最佳归并树算法”。基本思想是越短的文档列表越最早开始参与文档列表的求交过程，越长的文档列表越要推迟计算求交的过程。每次对现存的两个最短的文档列表进行归并，直到归并全部的文档列表，每次归并后减少一个文档列表。因此如果是 n 个文档列表，则需要归并 n 趟。下面通过一个例子来说明这个归并过程，最后得到的一个最佳归并树能够让我们完整地理解这个方法。

假定由多个查询词检索出如下的文档列表组，文档列表按照长度进行升序排序。

$doclist_1$(2,5,7,9,10,15,18)

$doclist_2$(1,2,3,7,8,9,10,12,15,18,22)

$doclist_3$(1,5,7,8,9,10,15,16,18,19,22,23)

$doclist_4$(4,5,6,7,8,9,10,11,15,16,17,18,19,20)

第 1 步归并最短的两个文档列表，即 $doclist_1$ 和 $doclist_2$。由于文档列表从倒排索引表中取出，因此文档列表中包含的文档自然地按照文档编号升序排序。这样归并两个有序的文档列表非常简单且高效，计算方法如下。

（1）为 $doclist_1$ 和 $doclist_2$ 分别设置一个当前下标 point_1 及 point_2 并初始化为 0，转（2）。

（2）如果 point_1 > len（$doclist_1$） 或者 point_2 > len（$doclist_1$），则求交结束；否则转（3）。

（3）如果 $doclist_1$[point_1]==$doclist_2$ [point_2]，转（4）；否则转（5）。

（4）如果 $doclist_1$[point_1]为一个交集元素，将其存放在返回结果中。point_1++，point_2++，并转（2）。

（5）如果 $doclist_1$[point_1]> $doclist_2$ [point_2]，point_2++；否则 point_1++，并转（2）。

读者不妨拿出纸笔用这个计算方法一步一步地将求 $doclist_1$ 和 $doclist_2$ 的交集过程画出来，实际的文档列表求交集实例第 1 步如图 5-4 所示。

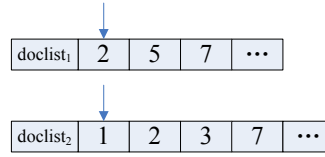


图 5-4　文档列表求交集实例第 1 步

首先初始化下标同时指向两个 doclist 的首个元素，通过比较得到 $doclist_2[0]$大于 $doclist_1[0]$，将指向小值的 $doclist_2$ 的下标向右移动一位，如图 5-5 所示。

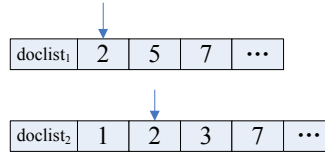


图 5-5　文档列表求交集实例第 2 步

当前 $doclist_1$ 和 $doclist_2$ 的下标指向的值相等，均为 2。因此 2 作为一个交集元素输出，并且将 $doclist_1$ 和 $doclist_2$ 的下标同时向右移动一位，如图 5-6 所示。

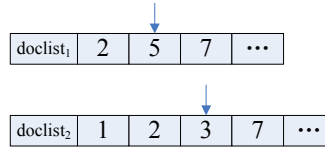


图 5-6　文档列表求交集实例第 3 步

此时 $doclist_1$ 的下标值为 5 大于 $doclist_2$ 的下标值 3，因此将指向小值的 $doclist_2$ 的下标向右移动一位，如图 5-7 所示。

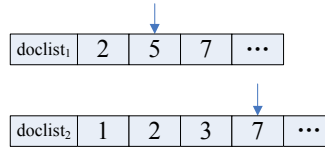


图 5-7　文档列表求交集实例第 4 步

接下来的计算将找到 7 作为一个交集元素输出，完成这个过程可以得到 $doclist_1$ 和 $doclist_2$ 的交集为（2，7，9，10，15）。

这种计算过程是线性的。因为每次无论何种情况，至少有一个文档列表的指针

（point_1 或 point_2）都向右移动一位。只要有一个文档列表指针移动到文档列表尾部，整个计算结束。

此外，很容易证明计算结果的交集长度必然比 $doclist_1$ 和 $doclist_2$ 都要短，至多和最短的文档列表长度相同（当 $doclist_2$ 包含 doclist_1，或者说 $doclist_1$ 是 $doclist_2$ 的一个子集）。由于文档列表组是按照文档列表长度升序排列的，$doclist_1$ 和 $doclist_2$ 的交集（不妨称为“doclist1_2”）下一次必定和 $doclist_3$ 求交，因为此时文档列表组中最短的两个文档列表必然是 doclist1_2 和 $doclist_3$。

不难得出，最佳归并的顺序如图 5-8 所示。

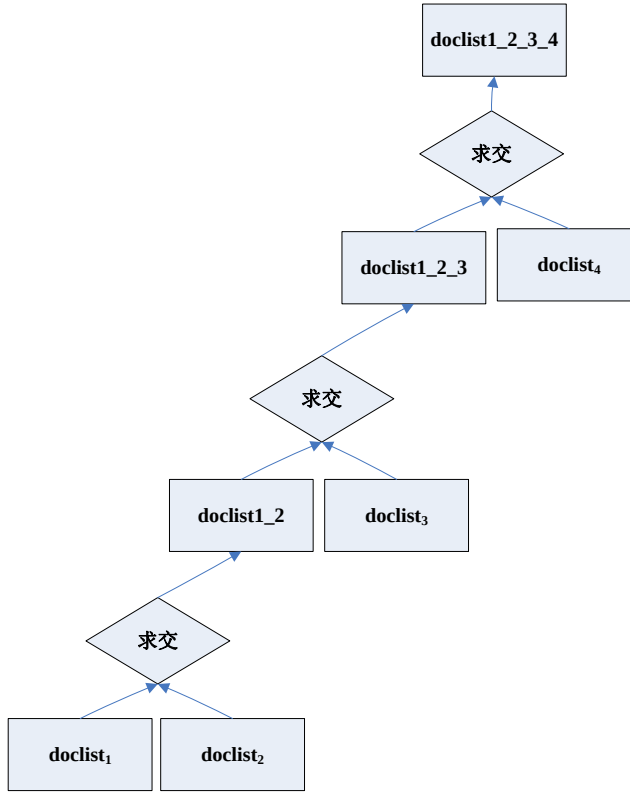


图 5-8　最佳归并顺序

最佳的归并顺序恰好就是按照文档列表长度排序的顺序，$doclist_1$ 和 $doclist_2$ 归并。归并结果和 $doclist_3$ 归并，归并结果再和 $doclist_4$ 归并。

这里值得一提的是，对于那些布尔逻辑表达式中带有“求非”的计算中，“求非”计算优先进行，并且和最短的求与运算的文档列表求非。举个例子，假定出现这样的查询词“搜索引擎 全文检索-检索”，那么这个查询翻译成自然语言为查询包含“搜索引擎”且包含“全文检索”，但不包含“检索”的文档（搜索引擎一般把减号作为去掉某种关键词的查询）。用“搜索引擎”关键词检索出的文档列表为 $doclist_1$，用“全文检索”关键词检索出的文档列表为 $doclist_2$，用“检索”关键词检索出的文档列表为 $doclist_3$。假定 $doclist_1$ 长度大于 $doclist_2$，则无论 $doclist_3$ 的长度如何，首先将 $doclist_2$ 里去掉和 $doclist_3$ 相同的部分，即 $doclist_2$-（$doclist_2 \cap doclist_3$）。这里的减号表示从 $doclist_2$ 中去掉 $doclist_2 \cap doclist_3$ 后剩余的部分。然后再与 $doclist_1$ 求交，这样完整的表达式为 $doclist_1 \cap$（$doclist_2$-（$doclist_2 \cap doclist_3$））。

这种求文档列表交集的方法的一个最大优点是如果在求最短的两个文档列表的交集时发现为空，即可终止整个求交过程，因为可以断定文档列表组的交集为空。例如，在如图 5-8 的的计算中，如果 $doclist_1$ 和 $doclist_2$ 的交集为空，则无须继续计算，这 4 个文档列表的求交集的结果必然为空。

当然也存在如下主要缺点。

（1）计算有依赖性，难以并发。计算串行完成，强行并发会导致大量空间浪费。

（2）需要在本地开辟额外的空间保存临时的求交结果，总的额外空间大小为第 1 次求得的交集长度。

最后，采用这种归并方法不可避免地存在这样的问题。即在最后一次归并时，必然是一个最短的文档列表和最长的文档列表求交集的过程。如前所述，最后一次归并时，doclist1_2_3 是 3 个文档列表的交集。必然小于或等于最短的文档列表 $doclist_1$，而 $doclist_4$ 是最大的文档列表，因此可能出现一个极小长度的文档列表和一个极大长度的文档列表求交。特别是查询词中包含一个低频词（例如“全文检索”）、一个高频词（例如“中国”）和多个中频词时，最后的结果不可避免地出现这种极小文档列表和极大文档列表求交的情况，从而给计算带来极大的麻烦。

关于文档列表求交的计算方法还有很多，各有优点和缺点。每种方法也包含了很多优化手段，这里不再展开。接下来将进入检索计算的最后一个计算环节——检索结果排序。

5.2.6 检索结果排序

由文档列表组求交集得到的每个文档都需要和查询词一同经过向量化的过程，通过计算文档向量和查询向量的夹角余弦求得向量相似度（一个可以量化的数值），排序就按照这个数的大小关系进行排列。由于搜索结果是海量的，用户也几乎不会耐着性子看完全部的检索结果。有调查表明，大部分的用户使用搜索引擎查询时，在得到搜索结果页后不会向下翻页，而只关注搜索结果的第 1 页。即只实际上需要返回前 *n* 项结果即可，学术上称此为“top-*n* 查询”。

由于文档列表按照 PageRank 排序（参阅前面的相关章节），这一点在索引系统中提到过（实际上，文档列表既按照 PageRank 排序，又按照文档编号排序），因此只需要 PageRank 排名靠前的一部分网页拿出来进行这种向量化，然后和查询词相似度的比较即可。而不需要把关键词 doclist 的全部文档都执行这样的计算，这样可以大大降低向量化和向量相似度计算的规模。

例如某个查询词通过布尔模型的求交过程得到 20 万个包含查询词的文档，这里假定只需要查询排名在 256(top-256)以前的结果，因此可以从 20 万个文档中取出一定比例的文档。例如取出前 5 000 个文档，注意由于这 20 万个文档是按照 PageRank 排序的，所以前 5 000 个文档可以理解为在这 20 万个文档中重要性最高的文档。接下来继续在 5 000 个文档中通过向量化及相似度计算，分别得到这 5 000 个文档和查询词的匹配程度。我们不妨在这里称为“匹配排名”（MatchRank），表示和查询词的匹配程度。例如使用堆排序或者快速排序这样的经典排序算法对 5 000 个文档的 MatchRank 进行排序，最终取排名前 256 位的文档即可。这样检索出的文档既具有重要性高（PageRank 高）的特点，也具有向量空间模型所要求的相关性强的特点。

在众多排序算法中，由于堆排序具有元素移动少、空间复杂度低并支持 top-*n* 查询等优点，因此被用来进行检索结果的排序。

5.2.7 堆排序

1991 年计算机先驱奖获得者、斯坦福大学计算机科学系教授罗伯特·弗洛伊德（Robert W．Floyd）和威廉姆斯（J．Williams）在 1964 年共同发明了著名的堆排序算法 HEAPSORT[Willioms 1964]。

首先给出堆排序定义，即大小为 n 的数组 A，下标依次为 1，2，…，n，称为“堆”，当且仅当该数组满足如下性质（简称为“堆性质”）：

（1）$A[i] \leqslant A[2i]$且 $A[i] \leqslant A[2i+1]$ 或（2） $A[i] \geqslant A[2i]$且 $Ai \geqslant A[2i+1]$

若将此数组看做是一棵完全二叉树的存储结构，则堆实质上是满足如下性质的完全二叉树，即树中任意一个非叶结点的关键字均不大于（或不小于）其左右孩子（若存在）结点的关键字。例如序列（10，15，56，25，30，70）和（70，56，30，25，15，10）分别满足堆性质（1）和（2），故它们均是堆。其中根结点（也称为“堆顶”）的关键字是堆中所有结点关键字中最小者的堆，称为“小根堆”（符合堆性质（1）），如图 5-9 所示；根结点（也称为“堆顶”）的关键字是堆中所有结点关键字中最大者，称为“大根堆”，它符合堆性质（2），如图 5-10 所示。

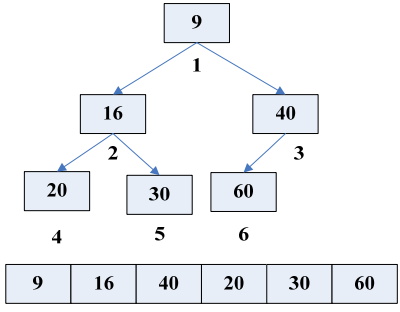


图 5-9　小根堆

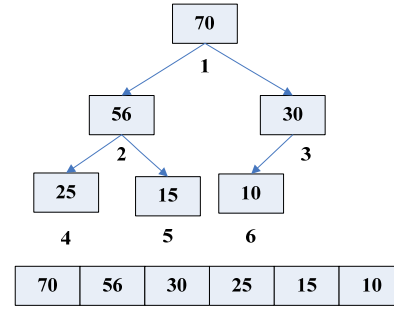


图 5-10　大根堆

接下来以大根堆（可以理解为符合大根堆性质的数组）为例，考察堆排序的过程。

（1）当前堆仅剩下堆顶（堆大小为 1）时堆排序结束，否则转（2）。

（2）堆顶和堆尾置换。

（3）将除堆尾以外的余下元素调整，并符合大根堆性质，转（1）。

不妨考察图 5-10 所示的大根堆的排序过程，堆顶和堆尾置换如图 5-11 所示。

首先将堆顶和堆尾置换（1 号位与 6 号位置换），这样 1 号位存放的元素为 10，6 号位存放的元素为 70，如图 5-11 所示。

接下来需要调整当前堆（注意，这里当前堆不包含 6 号位，即除 70 以外），以

符合大根堆的性质。

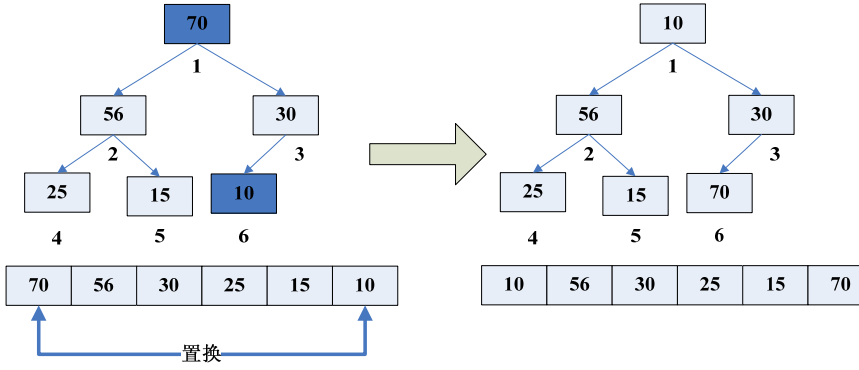


图 5-11　堆顶和堆尾置换

为调整大根堆，首先将堆顶 10 与其两个孩子 56 及 30 比较。由于 56 是最大值，因此将 10 与 56 置换。即置换 1 号位与 2 号位，如图 5-12 所示。

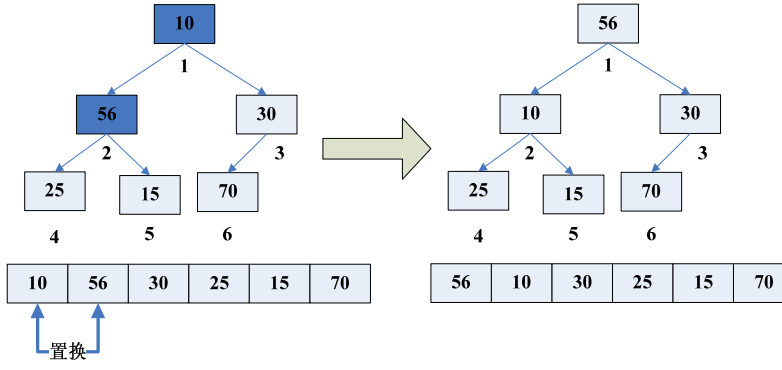


图 5-12　置换 1 号位和 2 号位

继续调整以 10 为堆顶的子堆（10，25，15），其他部分都已符合大根堆的性质，调整 10 为堆顶的子堆。

首先将堆顶 10 与其两个孩子（页结点）25 及 15 比较，由于 25 是最大值，因此将 10 与 25 置换。即将 2 号位和 4 号位进行置换，置换结果如图 5-13 所示。

注意第 1 趟的排序结果将最大值 70 排定顺序，第 1 趟排序后，当前堆为（56，25，30，10，15），依然符合大根堆的如下性质。

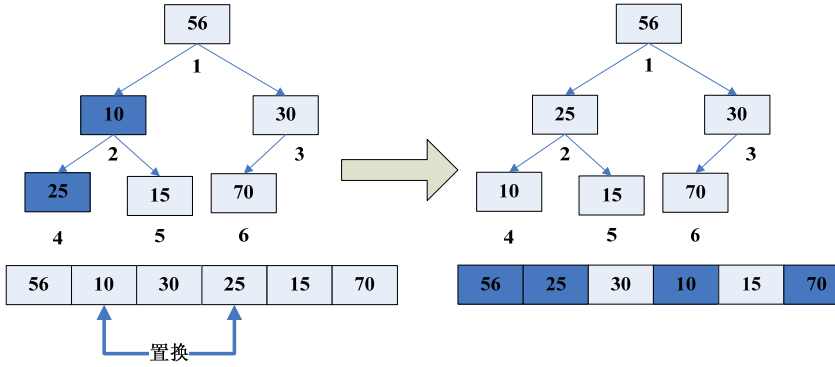


图 5-13　置换 2 号位和 4 号位

1 号位元素大于 2 号位和 3 号位元素（56≥25；56≥30）。

2 号位元素大于 4 号位和 5 号位元素（25≥10；25≥15）。

在第 1 趟排序中，发生移动过的部分如图 5-13 右图中深色块所表示，即 56，25，10，70。而 30 和 15 所在的块并没有发生变化，可见发生移动的数组元素是很少的。可以证明发生移动的元素个数为 0(lg*n*)数量级的，其中 *n* 表示当前堆的数目，lg*n* 为当前堆的高度。每次堆顶和堆尾的置换，或者调整成堆的置换过程至少需要 3 次内存复制（两个元素相互交换至少需要 3 次内存复制）。因此置换次数较少，节约了大量的内存复制。如果是这些元素存放在较大的文件中，则因为节省了 I/O 次数，使得在性能上带来的提升是十分显著的。综上所述，堆排序具有内存复制少的特点。

继续将当前堆的堆顶和当前堆的堆尾 5 号位置换（原堆尾为 6 号位，注意每次执行一个操作，当前堆尾位置减 1），调整成大根堆。周而复始，直到当前堆中只剩下堆顶时排序结束，如图 5-14 所示。

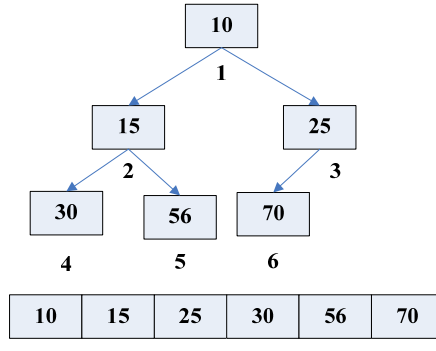


图 5-14　堆排序后的结果

除了具有内存复制少的特点，堆排序还具有“就地”排序（in-place）的特点。即排序过程只需要一个置换时的临时元素的存储开销，不需要额外的存储开销。因此堆排序的空间复杂度是 0(1)，这个特点可以归纳为空间占用少。

此外，堆排序特别有利于进行 top-*n* 的查询。即仅需要排序前 *n* 个元素，而不需要完全排序所有的元素。例如在如图 5-10 所示的例子中，如果只需要排序前两个元素，那么排序的方法可以改为：

（1）定义“已经排序的元素个数”变量 sorted_num++，并初始化为 0。

（2）当前堆仅剩下堆顶（堆大小为 1 时），堆排序结束；否则转（2）。

（3）堆顶和堆尾置换，sorted_num++。

（4）若 sorted_num==2，堆排序结束；否则转（5）。

（5）将除堆尾以外的余下元素调整，并调整当前堆使之符合大根堆性质，转（1）。

排序结束后，数组末位存放最大值，倒数第 2 位存放次大值（second-large），如图 5-15 所示。

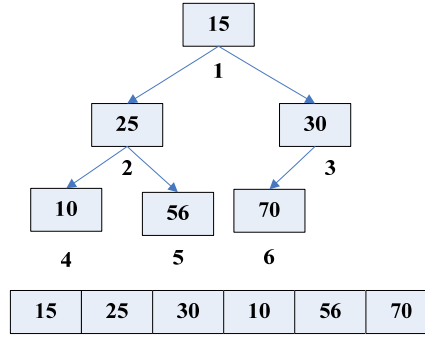


图 5-15　Top-2 排序后的结果

由于其他位置是否有序并不关心，所以排序可以在排好倒数第 2 位后停止。可见堆排序在仅排序前 *n* 个元素具有计算量低，内存复制少的性质。

综上所述，堆排序具有内存复制少、额外空间占用少并适合 top-*n* 的排序的特点，因此被用于查询词和文档相关度排序的计算中。

5.3　中文自动摘要

中文自动摘要是搜索引擎返回给用户查询结果的重要组成部分，使得用户只需要浏览摘要即可了解需要查询的信息。

5.3.1　自动摘要的发展历史

自动摘要技术比较复杂，涉及自然语言处理（NLP）的核心部分。在详细介绍这部分知识之前，首先来回顾自动摘要这项技术的发展历史。

1958 年，IBM 公司的 H.P.Luhn 首次发表第 1 篇有关自动生成文摘的文章[Luhn 1958]，宣告了该项技术的诞生，至今自动文本摘要的研究已走过了 40 多年的历史。

进入 90 年代，随着电子出版系统和互联网的蓬勃发展，自动文本摘要的价值充分显露出来，越来越受到国内外研究者的重视。

1993 年 12 月，在德国 Wadern 召开了历史上第 1 次以自动文本摘要（Summarizing Text for Intelligent Communication）为主题的国际研讨会。

1995 年，国际期刊 Information Processing & Management 出版了一期题为“Summarizing Text”的专刊，标志着自动文本摘要时代的到来。

1995 年以后，特别是以 Google 为代表的搜索引擎的兴起，自动摘要技术被应用到了搜索引擎的查询结果展示上。

以下我们所说的自动摘要特指搜索引擎领域内的自动摘要提取技术。

5.3.2　自动摘要的含义和实现

自动文本摘要简称“自动摘要”，是从文档中自动提取出的一个正文片断。用户仅仅需要浏览自动摘要就能够了解文档中与查询词相关的部分，进而判断是否值得详细阅读整篇文档。对于同样的一篇文档，其自动摘要对于不同的查询词是不同的。因此自动摘要的计算是实时的，并且是和查询相关的。既需要考虑“效率”，也需要考虑“效果”。

自动摘要在搜索引擎中的实际应用如图 5-16 所示。

图 5-16　摘要样式

其中方框标出的部分为自动摘要部分，可以看出自动摘要的内容均为实际网页中正文的一个片断。查询词用红色字体标出，这种标识位置信息的技巧在搜索引擎行业称为“标红”。

摘要是搜索结果重要的一个环节，从严格定义上说，它必须包含如下 4 层含义。

（1）**摘要指示性**：摘要必须出现查询词，必须能够指出查询词在文档中的位置。

（2）**摘要描述性**：如果是多个查询词，摘要有限的篇幅最好能够包含全部查询词。如果不能包含全部查询词，也需要尽可能包含权重更高的查询词。

（3）**摘要简洁性**：摘要长度必须控制在一定范围内，既不能太短，也不能太长。

（4）**摘要完整性**：摘要的句子必须完整，而且摘要的每个组成部分都必须从句子的首部开始，不允许中间断句。

结合第 3 章“搜索引擎分析系统”中提到的投票算法，以及这里介绍的滑动窗口方法可以较好地解决自动摘要的提取问题，并且满足上面提到的摘要的 4 个特性——指示性、描述性、简洁性和完整性。

滑动窗口实现自动摘要包括如下步骤。

（1）在文档正文中标记查询词出现的位置（这部分工作事实上在创建倒排索引时已经完成，每个关键词在文档中出现的位置均被标识）。

（2）从第 1 个查询词开始，取出窗口长度的正文片断作为第 1 个候选窗口。接下来，每次窗口滑动到下一个出现的查询词开始。同样取出窗口长度的正文片断作为候选窗口，直到取完全部的候选窗口。

（3）在每个候选窗口包含的正文片断中，累计候选窗口中出现的全部查询词的权重作为候选窗口的评分，最终评分高的候选窗口选做自动摘要提取的结果输出。

滑动窗口的方法与第 3 章中介绍的 Shingle 算法很类似，不过这里每次滑动的距离不定。如果大段的章节没有出现查询词，则能够一次跳过。由于降低了候选评分的计算量，因此提高了自动摘要计算的效率。

下面通过一个完整的例子来说明整个摘要提取的过程，假设有如下这样一个文档（用斜体表示）。

搜索引擎包含了各个学科的概念和知识，这些学科包含了计算科学、数学、心理学等。特别是数学几乎在搜索引擎的各个系统都大量使用，例如布尔代数、概率论、数理统计等，这些数学知识的应用为搜索引擎解决了一个个的难题，最终使得搜索技术走向成熟。

假设每个滑动窗口取 40 个汉字，标点符号也作为一个汉字。查询词为“搜索引擎 数学”，其权重采用 TF/IDF 方法量化后分别为 6 和 4，那么摘要提取的步骤分为下面几个阶段。

（1）计算查询词在正文中的位置信息，并由<位置，查询词长度，查询词权重>这样的三元组来表示，如图 5-17 所示。

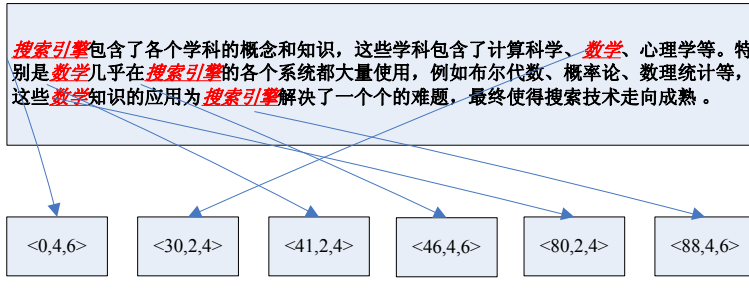


图 5-17　标记查询词在文档中的位置

在标记位置的 3 个分量中，第 1 个分量表示查询词在文档中的位置，第 2 个分量表示查询词的长度，第 3 个分量表示查询词的权重。例如第 1 个位置信息<0,4,6>表示的含义为在文档的位置 0 处出现了一个权重为 6 且长度为 4 个汉字的查询词。

（2）从文档正文第 1 个查询词出现的位置开始，取窗口长度大小的片断作为第 1 个候选窗口。下面每次窗口滑动到下一个查询词出现的位置，同样取窗口长度大小的片断作为下一个候选窗口。这样循环往复，直到取完所有的候选窗口为止。如果在上一步中标识了 *n* 个查询词出现的位置，按照这种计算方法，理论上就应该有 *n* 个候选窗口。参考图 5-17 的查询词位置标识（斜体下划线的词为查询词），可能的候选窗口的起始位置为 0、30、41、46、80 及 88。每个窗口最大取 40 个汉字，最后得到 6 个候选窗口如图 5-18 所示。

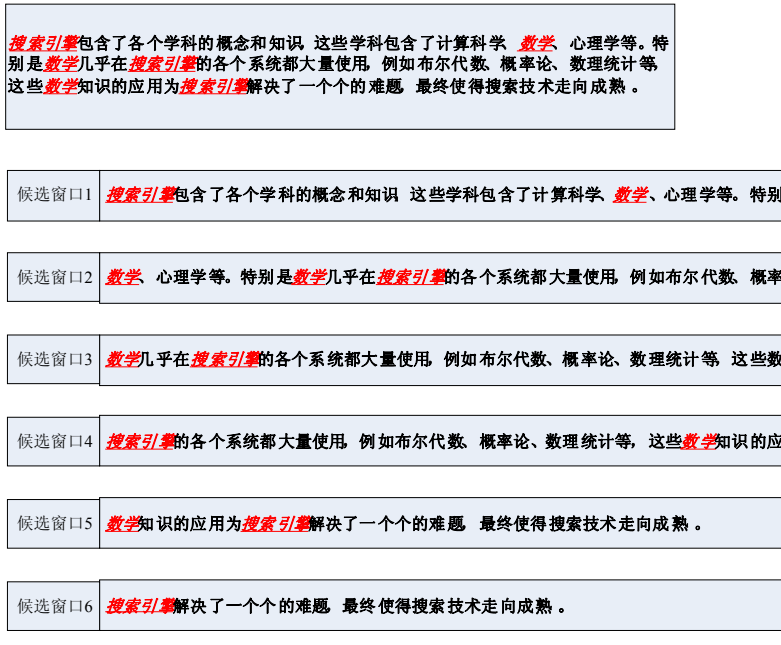


图 5-18　通过滑动得到的 6 个候选窗口

在图中，由于正文包含了 6 个关键词，因此从每个关键词开始都需要取一个窗口，或者理解为窗口每次滑动到下一个关键词开始处。这样共有 6 个候选窗口，每个窗口最大取 40 个汉字。

事实上，还需要设置最小窗口长度，用来保证摘要不会过短。如果最小窗口长度设置为 30，则窗口 6 就不会成为候选窗口，从而避免不必要的计算。

（3）利用投票算法对各个候选窗口打分，分数最高者为最佳摘要。

如下分别计算每个窗口的得分情况。

候选窗口 1：包含“搜索引擎”和“数学”各一次，因此得到 10 分。

候选窗口 2：包含“搜索引擎”一次，“数学”两次，因此得到 14 分。

候选窗口 3：包含“搜索引擎”和“数学”各一次，因此得到 10 分。

候选窗口 4：包含“搜索引擎”和“数学”各一次，因此得到 10 分。

候选窗口 5：包含“搜索引擎”和“数学”各一次，因此得到 10 分。

候选窗口 6：包含“搜索引擎”一次，因此得到 6 分。

综上，最佳摘要为候选窗口 2 包含的正文片断。回顾摘要包含的 4 个含义，考察候选窗口 2。虽然其中包含了查询词，也具备了一定的长度，而且通过投票算法打分胜出。然而候选窗口 2 表达的句子不够完整，至少没有从一个完整句子的句首开始。

在前面的例子中，假定窗口大小固定不可变，因此取出不完整句子是不可避免的，因此必须从窗口大小入手解决这个问题。即允许窗口大小在一定范围内变动，这种变动主要包含如下两种情况。

（1）窗口缺失句首部分，尽可能向前多包含几个汉字。例如候选窗口 2，缺少了句首“特别是”这个词，因此窗口可以增大头部包含这 3 个汉字。

（2）窗口包含了下个句子的开头部分，而并没有实际的意义。例如候选窗口 3 包含了下一个句子的开头部分“这些数”，这些部分可以去掉，因此窗口可以缩小尾部去掉这 3 个汉字。

综上，实际的候选窗口能够通过标点符号进行小范围内的调整，以尽可能地包含一个完整的句子。微调后的候选窗口如图 5-19 所示。

由于使用候选窗口的微调，所以可能会出现如图 5-19 所示的候选窗口重复的情况。例如，候选窗口 5 和候选窗口 6 包含完全一致的正文片断，这些相同的候选窗口需要合并成一个候选窗口。接下来同样采用投票打分的方法，可以得到候选窗口 2 是最佳候选窗口。因此对于“搜索引擎 数学”这样的查询，提取的自动摘要为正文片断“特别是数学几乎在搜索引擎的各个系统都大量使用，例如布尔代数、概率

论、数理统计等”。

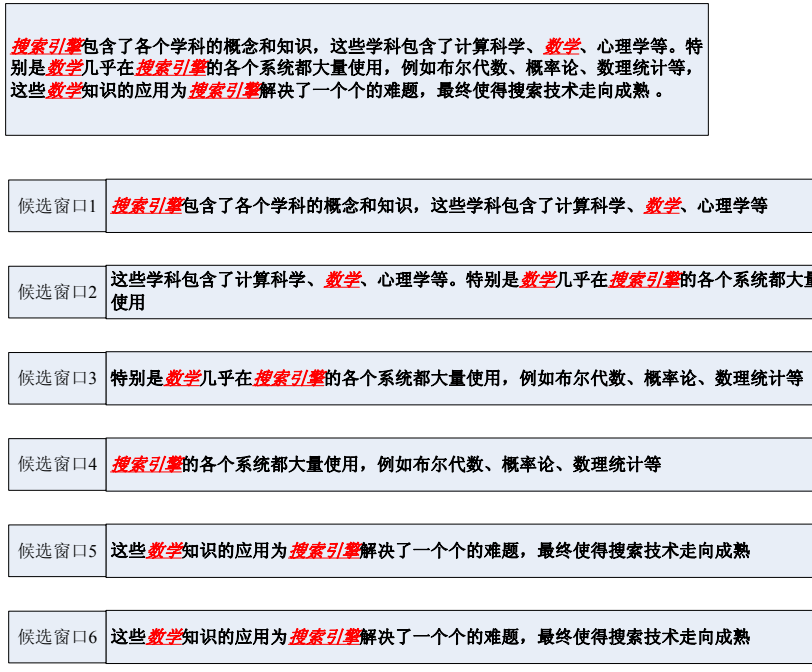


图 5-19　微调后的候选窗口

完成自动摘要的功能后，查询系统就可以拼接检索模块得到的文档和自动摘要模块得到的摘要，从而生成最终的搜索结果页，下面一节中将介绍如何生成搜索结果页。

5.4　生成搜索结果页

对搜索引擎的用户来说，搜索结果页是离其最近的部分。搜索结果页的主体包含与查询的相关网页链接（URL）和与查询相关的自动摘要（Automatic Summary），这两个部分的合成还需要一些额外的计算。

5.4.1　生成搜索结果页

第 4 章提到索引系统中局部倒排文件的分布式部署，如图 5-20 所示。

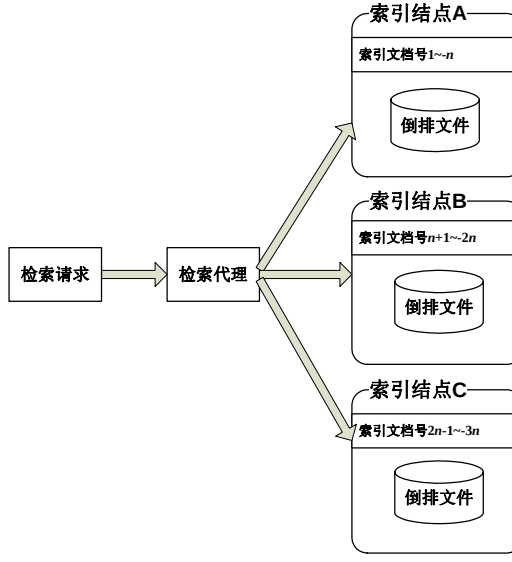


图 5-20　分布式部署

这种设计一方面实现了并发的检索，一方面提高了可靠性。正是由于索引结点的这种分布式部署，实际的检索也是在索引结点内部完成。每个索引结点增加一个检索模块从而变成了一个检索结点，一次检索请求引起的计算，直到最后的网页结果生成经历了如图 5-21 的过程。图中两处检索代理实际上是同一个系统，只是为了方便分开表示。

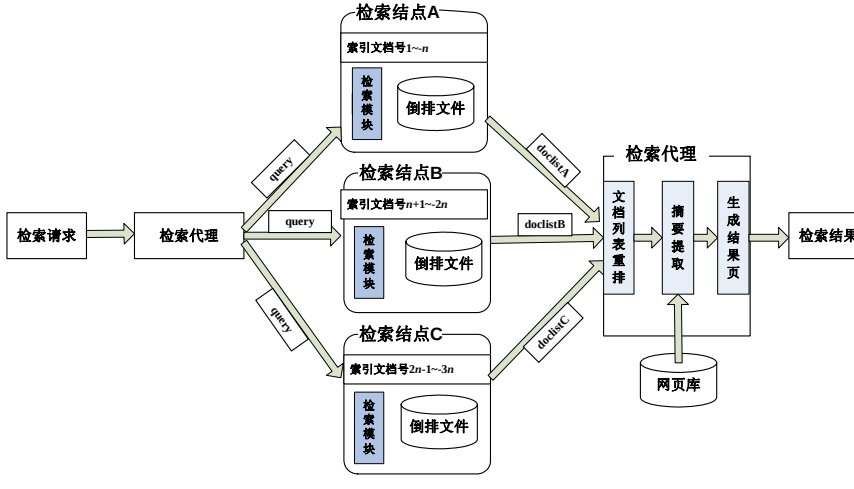


图 5-21　生成搜索结果页的全过程

在图 5-21 所示的搜索结果页生成全过程中主要经历了以下几个步骤。

（1）检索请求发给检索代理，检索代理进行查询词分词。

（2）查询词分词后的结果（图中用 query 表示）同时发往各个检索结点。注意这里的检索结点因为具有内部检索功能，因此和图 5-20 中的索引结点不完全相同。通过检索模块的计算，各个检索结点将各自本地倒排文件中检索出的文档列表发给检索代理（不同的索引结点返回不同的文档列表，分别用 doclistA、doclistB 和 doclistC 表示）。

（3）检索代理重新排序来自各个检索结点的文档，取出排名靠前的 *n* 个结果文档作为结果页拼接的候选文档。

（4）通过自动摘要模块从网页库中取出这 *n* 个文档的摘要信息。

（5）将（3）和（4）的结果合并，动态生成搜索结果页。

计算出一个搜索结果页需要历经如此复杂的步骤和操作，一个在线的搜索引擎每秒都需要响应相当多的检索请求。如果每次检索请求都经历这样的步骤，显然是不够经济的。与操作系统的缓存设计一样，搜索引擎也为搜索结果页设计缓存。用来减少重复计算，提高效率，下一节将讨论关于搜索引擎结果页缓存的一些设计。

5.5 搜索结果页的缓存

在查询系统中，搜索结果页的缓存（Cache）是对搜索“效率”贡献最大的设计。由于缓存中的搜索结果页都是前人查询的结果，因此用户的查询请求如果在缓存中命中（和前人的查询相同），则查询系统直接把缓存中存放的搜索结果页返回给用户。

用户在使用搜索引擎进行检索时，查询词可能千差万别。但是如果从大量用户的查询统计上看，总会有一些词汇经常被查询，有些词汇却很少被查询。文献[王建勇, et al.,2001]提出了如下一些结论。

（1）前 20 %的查询词的查询次数约占了总查询次数的 80%。

（2）查询具有稳定性，查过的词很可能在不久的将来还会被查询。

搜索结果缓存的实现方法和操作系统中提到的 LRU 算法基本一致，下面一起回顾一下 LRU 缓存置换算法。

回顾第 2 章中提到的网页库设计，对搜索结果页的缓存库必须能够支持随机访问，这一点很重要。如何支持这种随机访问其内部原理和数据库设计很相似，这里不再展开，有兴趣的读者可以参考 B+树等这类能够支持随机访问的索引方式。

有了搜索结果页缓存的设计，搜索引擎查询层就能够大大降低重复的计算量，提高同时响应用户检索请求的能力。具有搜索结果页缓存功能支持的查询系统如图 5-22 所示。

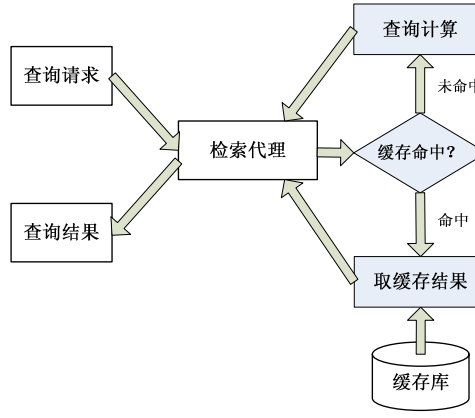


图 5-22　具有搜索结果页缓存功能支持的查询系统

增加了缓存功能后，查询系统可以较少执行实际的查询计算，而采用重用缓存中保存的历史相同的查询结果网页的方法来大大提高查询效率。目前的技术能够达到在缓存中命中 99%的查询，因此用户实际的查询绝大多数情况都是取自缓存的搜索结果页，这就是搜索引擎为什么能够如此快速地返回查询结果的一个重要原因。

也许是由于搜索结果页缓存的出色设计，在“效率”和“效果”之间的竞争上，“效率”占据了优势。因此近年来，查询系统的研究方向主要在“效果”上，而效果的追求还需要推测用户的查询意图。如果能正确地推测出用户的查询意图，那么对效果的改善可以说是大大有利的。下一节中我们将了解一些关于搜索引擎推测用户查询意图的思想和方法。

5.6　推测用户查询意图

推测用户查询意图首先需要对用户的查询做出基本分类，然后通过日志分析及

挖掘的技巧对排名进行干预，最终通过大量用户的行为引导最终的排名结果。

5.6.1 查询分类

在实际应用中，用户检索的需求多种多样。根据 Broder 等人对 Alta Vista 搜索引擎的用户日志分析工作，检索可以根据查找信息目的不同分为 3 类，即导航类查询、信息类查询和事务类查询。

2004 年，Yahoo 公司的 Danny 等人在 Broder 工作的基础上将检索类型进行了细化。但总体分类结构则基本保持不变，这也说明了这种查询分类体制的可靠性。

导航类查询的目标是查找一个用户已知的网页（帮助其找到对应的 URL），例如，“软件学报主页”及“清华大学招生简章”都属于导航类检索的范畴。导航类检索按照查找目标页面的不同细分为特殊需求页面定位任务和主页定位任务两类，主页定位任务的目标页面是站点/子站点的主页；而特殊需求页面定位任务的目标页面则是主页以外的页面。

信息类检索的目标是查找关于某个查询主题的相关信息，如“加强党的执政能力”就可以算做信息类查询。

事务类检索则是用于查找关于某个内容的网络服务，如购物服务、查询服务及下载服务等，典型的样例如“mp3 下载”等。

在这 3 类查询中，不同的查询有着本质的不同。

导航类查询，例如主页的查询可以充分利用锚文本、关键词的位置信息（标题或正文具有不同的价值）及 PageRank 等信息进行查询，总体查询效果是十分理想的。目前绝大多数的搜索引擎都能对这样的查询做到“首条命中”，比如查询“南京大学”，首条必然是南京大学的主页。

而信息类和事物类查询效果就目前来说，和导航类查询差距很大。例如查询“如何买南京到北京的返程票”及“Z50”（南京到北京的一个火车车次），有时搜索引擎返回的查询首页中，10 个查询结果中只有 2~3 个是有效的。因此对于查询效果来说，主要的难点在于解决信息和事物类的查询，目前各大搜索引擎都对这类查询效果表现出了极大的关注。

5.6.2　推测信息类、事物类的查询意图

导航类查询通过去除搜索结果中低质量的网页或者排名靠后的方法来提高查询效果，而信息和事务类查询是将搜索结果中高质量的网页排名靠前的方法来提高查询效果，而主要的难点在于对于信息和事务类查询。例如用户查询“Z50”，可能是一个列车的车次，也可能是一款手机的型号，或者是一款数码相机的型号。那么用户查询的到底是什么呢？在推测用户搜索意图上，查询系统做了如下的工作。

（1）从查询日志中得到用户的这类查询中实际点击的 URL，并进行排名反馈。

如果在 100 个查询“Z50”的用户中，50 个选择了查看列车车次的 URL，10 人选择了查看数码相机 Z50 的 URL，5 人选择了查看手机 Z50 的 URL，由此反馈给排名系统。在下次再有类似的查询，将和列车车次有关的 URL 排第一，依次为与数码相机 Z50 相关的 URL，以及与手机 Z50 相关的 URL。这种利用用户的实际点击来进行排名可以满足主流查询用户的需求。

（2）在用户的查询序列中分析查询意图，并给出搜索提示（Query Suggestion）。

通过对用户查询日志的统计分析，例如 10%的用户在查询“Z50”后，继续查询了“火车 Z50”或者“Z50 车次信息”，那么可以推测用户在查询 Z50 后并不满意给出的结果，而使用了更加精确的查询词。因此在搜索提示（一般出现在搜索页的尾部）中给出搜索提示，帮助用户选择更好的查询词得到搜索效果，同时将在查询“Z50”后的搜索结果中涉及火车车次信息的结果排名靠前。

综上所述，这类信息和事务类查询大多通过事后分析及日志挖掘的技巧将分析结果反馈给排名系统，使得在接下来的排名更加科学。因为具有用户的点击反馈和查询反馈，所以一个搜索引擎的用户越多，其查询效果就越好。

5.7　查询系统的当前热点和发展方向

搜索效果是搜索引擎的命脉，而改善搜索效果的主要途径为查询系统。因此查询系统是搜索引擎中最为热门的话题，目前已成为各大互联网主流会议的主要议题。

5.7.1 查询系统的当前热点

搜索引擎的查询技术发展到当前的水平，主要的发展方向可以归纳为以下几个方面。

（1）推测用户查询意图，这方面主要的工作包括查询纠错（Query correction）、查询推荐（Query Suggestion）及相关搜索等。

（2）能够在某个细分领域进行查询，例如查询行业信息，或者某个领域信息等，这方面工作主要包括垂直搜索及分类搜索等。

（3）查询结果的优化，这方面工作包括相似结果的聚类、垃圾网页及病毒网页的甄别等。

（4）提供个性化服务，保存用户的个性化信息并给出可定制的服务，例如可定制搜索服务等。

查询系统，特别是在理解用户查询词、理解用户本身及理解文档方面还有很大潜力可挖。例如向量空间模型还比较初级，没有达到语义分析的水平。再如查询“微机 行情”，文档中只包含了“电脑 行情”的有关信息。由于无法分析“微机”和“电脑”是否是一个概念，因此传统的向量空间模型无法解决这样的问题。

理解查询词、理解用户和理解文档在研究领域取得了一些突破，但是由于计算复杂等原因而难以产品化，离业界的需求还有一段距离。

5.7.2 查询系统的发展方向

搜索引擎的查询技术发展到当前的水平，主要的发展方向可以归纳为以下几个方面。

（1）推测用户查询意图，这方面主要的工作包括查询纠错（Query correction）、查询推荐（Query Suggestion）及相关搜索等。

（2）能够在某个细分领域进行查询，例如查询行业信息，或者某个领域信息等，这方面工作主要包括垂直搜索及分类搜索等。

（3）查询结果的优化，这方面工作包括相似结果的聚类、垃圾网页及病毒网页

的甄别等。

这些方向都取得了很大成绩，但还没有形成十分成熟的技术。这里不再展开介绍，有兴趣的读者可以查阅相关文献进一步了解这方面的研究成果。

参考文献

[Baeza-Yates and Ribeiro-Neto,1999] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*：Addison-Wesley- Longman, 1999.

[Luhn 1958]P H Luhn． Autornatic creation of literature abstracts． IBM Joumal，1958，2（4）：159~165.

[S. Brin 1998] S. Brin and L. Page. The anatomy of a large-scale hyper textual Web search engine. Computer Networks and ISDN Systems, 30（1-7）：107-117，1998.

[Shannon,1948] C.E.Shannon. A mathematical theory of communication. Bell System Technical Journal,27：398-403，1948.

[Willioms 1964]Willioms J W.Algorithm 232 "Heap sort"comm. ACM 1964（7）：347.

[王建勇, et al.,2001] 王建勇，单松巍，雷鸣，谢正茂，李晓明“海量 web 搜索引擎系统中用户行为的分布特征及其启示”中国科学 E 辑，vol. 31，No.4, pp. 372-384，2001.

[李晓明 2004] 李晓明，闫宏飞，王继民，“搜索引擎——原理、技术与系统”，科学出版社.

第 6 章　搜索引擎日志分析

6.1　简介

搜索引擎既然是一种用来在万维网上检索各种文件的系统，它的存在就离不开使用它的检索者。一个成规模的搜索引擎，一天内的用户数有成百上千万，甚至上亿，而他们与搜索引擎之间的交互次数更是十倍于此。如此纷繁复杂的交互过程背后，隐含了用户怎样的搜索需求？这些搜索的需求是如何被响应？最终的搜索结果用户是否认可？认可的程度如何？……弄清楚这些问题无疑对改进搜索效果有着极为重要的意义。而想要回答它们，就要在搜索引擎的日志中探寻答案。为什么这么说呢？我们还是从了解搜索引擎日志所包含的内容开始吧。

6.1.1　人机交互的记录——日志

日志，想必大家都知道是什么。平时写日记，或者在网络上写 Blog，用来记录自己的心情、感受或者是做了些什么事，也算是日志的一种。具体到计算机系统而言，日志指的是系统接收和响应的各种消息的记录，同时也指记录这些消息的文件。很显然，这样的记录是不带感情色彩和倾向性的。

搜索引擎，作为一种复杂的计算机系统，其日志大致可以分为系统日志和搜索日志两大类。前者包含诸如系统中硬件、软件运行的状态信息等，例如各时间段解析的数据条数记录，或者是返回一次查询的时间等。而后者，搜索日志，则记录着用户使用搜索引擎的过程，如用户搜索了什么，点击了哪些结果等。系统日志对于监控和改善搜索系统的性能有着重要的指导意义。本章要讲述的搜索引擎日志分析，将集中在第二类日志的分析上，着重介绍利用搜索日志改进搜索效果、提升用户体验这一方面。这类日志对于改进搜索系统的性能同样有重要作用，但这超出本书所打算讨论的内容范围，感兴趣的读者可以参考 Fabrizio Silvestri 和 Ricardo Baeza-Yates 在一系列搜索领域学术会议上所作的相关报告，例如 Query Log Analysis for Enhancing Web Search 等，也可以到这两位专家的个人主页上去查看更多的相关内容，并以此为起点进行扩展学习，这里不作更多的阐述。

不妨以一次普通的搜索过程为例，来看看搜索引擎都做了哪些记录，以此作为讨论搜索日志分析的起点。搜索过程从用户打开搜索引擎入口开始，用户首先选择搜索产品，是网页搜索还是其他类型的搜索。然后用户输入查询词，启动一次查询。

在返回查询结果后，用户会浏览结果页面，并点击认为有可能达成搜索目标的结果。当感觉搜索结果不佳时，用户会更换或修改查询词再进行查询，或者直接离开搜索引擎，从而完成一次完整的搜索任务。

在这个过程中，搜索日志中会记录用户到达搜索引擎的时间、来源站点，以及用户的 IP 地址，首选的搜索产品，各次输入的查询词，查询词是键入还是从推荐的查询词中选取的，对搜索引擎返回的查询结果，点击的是哪个结果，点击了哪些结果过滤或者排序选项，在结果页有没有切换搜索产品，以及这些动作发生的时间等。

搜索行为和产生日志的整个关系如图 6-1 所示。

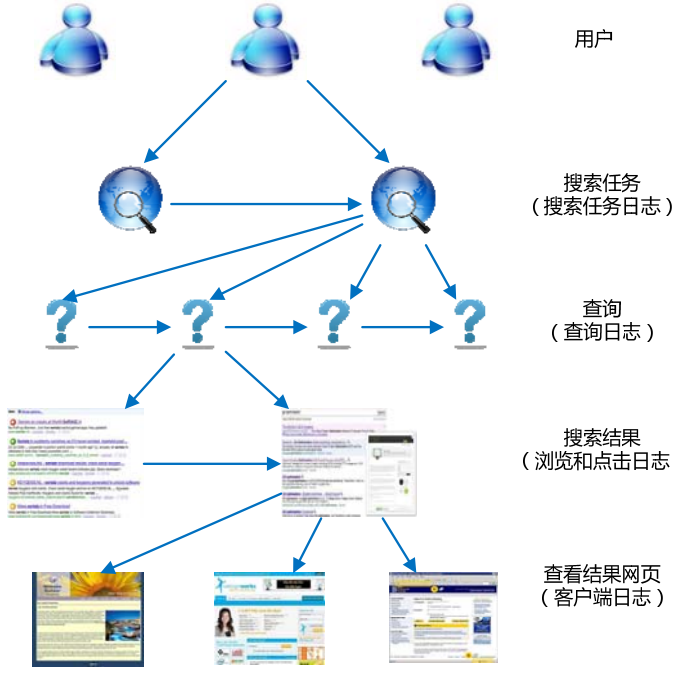


图 6-1　搜索行为和相关日志

由于用户发起的搜索行为，是通过搜索引擎的客户端（浏览器）发送到搜索的服务器端的，所以整个日志，依据产生日志的行为发生的位置，也可以分为服务器端日志和客户端日志两种。服务器端主要记录的信息有：

- 用户相关信息，如用户 ID、用户的来源 IP 等；
- 查询信息，如查询串、查询时间、查询终端类别（桌面设备、手持设

备）等；

- 点击信息，如点击的结果 URL、点击发生时间、点击的结果在排序中的位置等；
- 结果展现信息，如展现的搜索结果、广告结果、展现的相关查询串等。

而客户端所包含的信息则主要有：用户的 ID、IP 信息，展现给用户的结果信息，用户点击结果的顺序及时间等信息。

上述是比较基本的服务器端和客户端搜索日志的内容，稍作整理就能发现，这些日志有不少内容是相同或相近的，总体来讲，按其中所包含的信息，基本上可以分为以下两类：一类是查询日志，即和用户的查询行为相关的日志，包含查询的来源站点、来源 IP 地址、查询时间、查询词等；另一类则是点击日志，即和用户的点击行为相关的日志，包括用户点击结果的顺序、点击的结果展示选项等。此外还包括各次用户行为的来源 IP，用户所使用的浏览器，以及用户开始搜索行为的入口等，但前两类则是搜索日志中的主体部分。

6.1.2　分析搜索引擎日志的意义

单个来看，每条搜索日志都是平淡无奇的，但是搜集每个用户、每次查询、每次应答的日志，将它们放在一起，就会成为一个内含丰富、深藏玄机的资料集。例如将一个用户在一个时间段内，针对某个搜索目标所进行的一系列查询和点击记录放在一块，就可以大致推测，对这样的用户、这样的需求，哪些结果是他们希望看到的，从而优化结果的排序。此外，综合众多用户的查询日志，可以知道哪些查询词是用户搜得最多的，哪些查询词是某一段时间内搜索量增长最快的，从而分析搜索的趋势。大家对百度风云榜（http://top.baidu.com/）应该不会感到陌生吧，某个时期，大家最为关心的、搜索频率最高的事件，都在榜单里有所体现。这个榜单的生成就离不开千千万万平凡的搜索日志。同样，Google 也有类似的资料发布在互联网上，大家可以在 http://www.google.com/intl/en/press/zeitgeist/index.html 看到相关的内容，顺便领略一下，在 Google 的用户群中，大家关心的是什么内容。

这些只是日志中所体现的非常基本的内在信息。对搜索引擎日志的分析，还能帮助我们做理解用户的查询意图、理解搜索结果的内容、评判搜索结果质量、改进搜索系统等一系列有用的事情，可以为人们不断改进获取有效信息的便捷性提供有

力的帮助。

那么，搜索日志为什么具有这般非凡的能力？

大家可以带上这个问题，在本章的阅读中思考它，在今后的实践中反复体会它，不断给出你自己新的答案。也许每个人的答案都不一样，而且可能永远也没有最正确的答案，但有一个因素，在答案中是不可或缺的，那就是人的因素。这个因素的存在，使得很多问题有了不同凡响的解决方式。举个例子，众所周知的 PageRank 算法，在这个 Google 赖以起家的算法中就包含了这个神奇的因素。不论这个算法在数值计算和高维矩阵处理方面的设计如何精巧，就它存在的基石而言，网页的外链最初都是人为设置的。

网页开发者将他们对互联网上资源的认知体现在了网页的外链中。千千万万互联网用户不自觉的外链行为，造就了一个极复杂的网页外链关系网，从而为 PageRank 的诞生准备了最基本的条件。而 PageRank 又为人们在浩瀚的互联网中寻找有用的信息，建立了一种序关系。可能 PageRank 并不完美，但无愧是一次天才的尝试。某一个人的行为是微不足道的，但千千万万的用户，却可以使互联网具有神奇的性质。这就是人和网的关系，多么简单而深刻。

随着信息化和网络时代的进步，分析搜索日志，不仅有助于研究和理解用户与搜索系统的交互，其意义更在于理解人类与信息化环境的交互模式，探索信息时代中获取和利用海量信息的方法。正是因为如此，从工业界到学术界，为数众多的工程人员、研究人员倾注了极大热情来投身这一领域，从而使得该领域不断有新的思想和理念涌现，处在一个方兴未艾的时机。

那么搜索日志中究竟有哪些值得关注和探究的内容，人们又是如何利用这些信息来改善自身所处的互联网环境呢？这正是本章所要探讨的主题，我们将带着探索和求知的心，来了解这个领域，领略它迷人的风光。

6.1.3 本章的主要内容

搜索日志分析究竟包含哪些内容，工作应该分为几类，目前学术界还没有定论。按日志的处理流程而言，可以分为日志的搜集、日志预处理、日志分析等内容[Jansen 2006]。按分析所希望达到的目的又可以分为查询理解、文档理解、查询与文档匹配及用户理解等[Jiang 2010]。本章将采用类似于第二类的方法，依据用户与搜索系统

的交互流程来介绍日志分析的基本方面，并根据对日志分析所要达到的目的来展开内容。

用户与搜索系统的一次交互有两个基本步骤。首先，是用户发起查询，而后，用户对系统返回的结果进行浏览和点选。通过一次或多次反复交互，用户完成一次搜索。按这样的流程，本章也从查询日志的分析开始，介绍查询频率统计、查询分类等内容；而后介绍结果的返回和用户的点击日志分析，包括搜索结果的统计分类、结果重排序等内容；最后简要介绍日志分析所涉及的用户隐私问题。

整体来讲，搜索引擎日志中蕴涵的信息之丰富，以现在的分析理论、方法和手段，可能还远未能达到完全洞悉其奥秘的程度。如果说日志中所包含的信息是深深的蓝色海洋，那么现在我们所看到的还只是它浅浅的海湾。本章所要介绍的内容只是目前已有理论和实践中的一部分，目的在于让读者知道这个领域是做什么的，有什么基本的方法和原理，为大家了解这个领域作个铺垫，希望这些内容能引起大家的兴趣，引发大家的思考。

6.2　知识准备

了解搜索引擎日志分析，不妨从介绍日志分析中所常见的术语开始。这些术语之所以会被经常提及，是因为它们涉及搜索引擎日志分析中基本也是核心的内容，理解它们可以起到窥一斑而见全豹的作用。

6.2.1　二分图模型（Bipartite Model）

二分图，大家学习图论时应该已经有所了解，它又称作二部图，是图论中的一种特殊模型。设 $G=(V,E)$是一个无向图，如果顶点 V 可分割为两个互不相交的子集 (A,B)，并且图中的每条边（i，j）所关联的两个顶点 i 和 j 分别属于这两个不同的顶点集$(i \text{ in } A, j \text{ in } B)$，则称图 G 为一个二分图。

这个模型之所以在搜索日志分析中经常被提及，其原因在于，它具有表达和抽象两类不同事物之间对应关系的能力。举个例子，如图 6-2 所示，可以用二分图来表示查询和结果 URL 之间的对应关系。

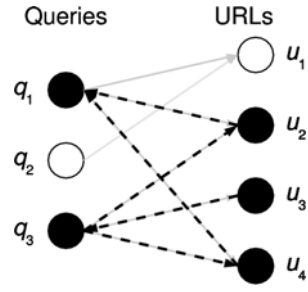


图 6-2　查询和结果之间的二分图关系模型

对于这种基本的二分图，如果给其条加上权重，就成为加权二分图。还是以上述查询和结果间的二分图为例，如果对于某个查询 q，和它所对应的某个结果 u 之间的边（q, u），加上查询 q 时，用户点击 u 的次数，那么所给出的二分图就是一个加权二分图了。

类似地，还可以建立用户与查询词之间的关系图。基于这两个关系图，更可以给出用户与结果之间的关系图。这样，依靠二分图就可以进行从用户到最终结果的一系列二元关系建模，可以进行诸如用户对哪些类别的查询或者是结果有偏好之类的分析。

6.2.2　图模型（graphical model）

日志分析中的图模型是指用图来表示概率分布的概率模型。其基本模式是把随机变量间的条件独立用图的形式表达出来，把概率分布表示为各个相关因子的乘积，以进行概率计算。

图模型主要有两大类，一类是贝叶斯网络（又称有向图模型）；另一类是马尔可夫随机场（又称无向图模型）。

其中有向图模型是通过一个有向图来表示一个概率分布，从而可以利用这个有向图模型来进行推断。对于一个概率分布 $p(x1,x2,\cdots,xn)$，依照概率论的方法，可以把它写成因子相乘的形式：

$$p(x1,x2,\cdots,xn) = p(x1)p(\mathrm{x2}|\mathrm{x1})p(x3|x1,x2)....p(xn|x1,x2....x(n\text{-}1))$$

当其中的某些随机变量是独立的或者条件独立的，就可以把上述式子进一步简化，例如 $x3, x1$ 在给定 $x2$ 的条件下是独立的，那么 $p(x3|x1,x2) = p(x3|x2)$。如果令每

个随机变量对应一个图的节点，从它的条件部分的每个随机变量节点连一条边指向非条件变量节点，就可以形成一个有向图模型。以概率分布 $p(x1,x2,x3) = p(x1)p(x2|x1)p(x3|x1)$为例，它的有向图模型可以表示为如图 6-3 所示。

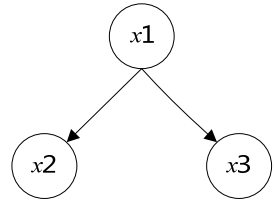


图 6-3　概率图模型

给定一个有向图，也可以得到这个图表示的概率分布。用图模型表示一个概率分布，其图形结构代表着各个因子的相互独立关系，另一方面，概率分布情况则可以用各个结点间的转移概率来表示。

无向图模型也是将变量间的条件独立用图表示的，从而使得概率分布成可以用因子乘积的形式，只不过它所采用的是无向图基础。其两个核心概念分别从以下两方面来理解。

（1）马尔可夫性：指的是一个随机变量序列按时间先后关系依次排开的时候，第 N+1 时刻的分布特性，与 N 时刻以前的随机变量的取值无关。拿天气来打比方，假定天气的变化过程是马尔可夫的，其意思就是我们假设今天的天气仅仅与昨天的天气存在概率上的关联，而与前天及前天以前的天气没有关系。

（2）随机场：当给每一个位置中按照某种分布随机赋予相空间的一个值之后，其全体就叫做随机场。其中有两个概念，位置（site）和相空间（phase space）。以种地来打比方，“位置”好比是田地，“相空间”好比是种的各种庄稼。给不同的地种上不同的庄稼，这就好比给随机场的每个“位置”，赋予相空间里不同的值。

马尔可夫随机场的基本思想就是：以当前各个结点状态向下一步状态的转变概率来描述概率分布，而用各个结点间是否有边相连来反应条件间的独立性。即有边相连的点，才能在一次状态变化之中相互影响。

无论有向图模型还是无向图模型，它们的结构都表明了因素之间的条件独立性，而它们的条件概率表或是状态转移概率则反映了概率分布情况。

日志分析中之所以会用到图模型，其原因在于采用类似二分图的离散方式来描

述用户、查询、结果之间的关系，那么不同元素之间的相互联系就可以依据统计值来进行基于概率的表达，而用图模型表达和分析这些离散元素间的概率关系则有直观自然的优势。下面就介绍一种具体的图模型——LDA，以加深了解。

6.2.3 LDA（Latent Dirichlet Allocation）模型

LDA[Blei 2003]模型可视作图模型的一种，它不同于 SVM 直接从样本得到判别模型的方法，而是由样本生成样本自身所属的概率模型，再依据样本自身的概率模型进行分类判别，类似于贝叶斯模型。该方法现在较广泛地应用于文本分析，主题发现等领域。

当仅考虑文本的词频，而不考虑词在文本中所处的位置和次序时，设文本中有 N 个单词，词表的规模为 V，同时文本含 k 个主题。

那么仅仅考虑词频概率时，推算文本一个词出现的概率，可以通过统计训练集中各个词的词频，得到一个基本模型。w 是离散随机变量，在词汇表 V 中取$|V|$个离散值。该模型中，一个词出现的概率也就是 $p(w)$，可视为训练集的词频。如果再引入文本主题，令离散随机变量 z，在主题集 T 中可取 k 个离散值，$p(z)$是 z 的分布。那么，一个词出现的概率则变为 $p(w|z)$。这一分布同样可以由训练集得到，这样，就将简单的词频归约到了各个主题下，变成了各主题相关的词频。

在生成某文本时，可以根据分布 $p(z)$选取该文本的主题，而后再根据分布 p(w|z)，选出相应的词。即可生成文本中的所有单词。这个方法假设一篇文本只有一个主题，词汇也仅局限于这个主题的词表之下。再扩展一步，由于文本可能包含多个主题，如用 Dirichlet 分布它描述文档到主题这一层面主题集的概率分布，即为 LDA 模型。

如图 6-4 所示，依据 LDA 模型，文本生成可用以下过程来描述：以 Dirichlet 分布 $p(\theta)$描述文档中主题集分布；然后再根据分布 $p(z|\theta)$选取主题 z_1，再根据分布 $p(w|z)$选出词集中的词 w_1。以此反复，直至选出 z_N，w_N，生成文本中的所有词。在不考虑词的位置和顺序的前提下，这一过程就生成了整篇文本。大家从该图中可以看到箭头表示条件概率，圆圈表示一个随机变量，观察到的随机变量用实心圈，而需要估计的则用空心圈。这样条件概率对应的贝叶斯网络即可用有向图表出。当随机变量及条件概率个数众多时，如果条件关系是类似的，则可用方框标识范围，而在框内记下这引起条件关系的个数，如图 6-4 中的 N 和 M。

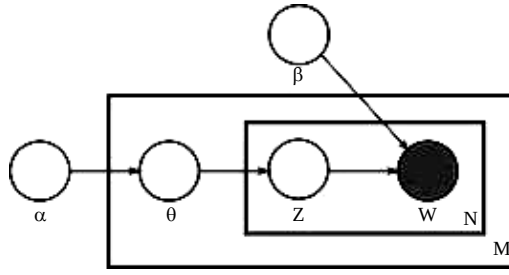


图 6-4　LDA 模型

在实际应用中，分布 $p(\theta)$由参数向量 $\boldsymbol{\alpha}$ 表达，而 $p(w|z)$则由参数矩阵 $\boldsymbol{\beta}$ 表达。从训练样本学习出 $\boldsymbol{\alpha}$ 和 $\boldsymbol{\beta}$ 两个参数，就能得到实际 LDA 模型的形式。具体可以采用极大似然法来估计。但由于 θ 和 z 为隐藏变量，所以似然函数不能直接计算。把 w 看做观察变量，θ 和 z 看做隐藏变量，采用 Expectation-Maximization (*EM*) 方法来处理。具体到这里，给定一组 $\boldsymbol{\alpha}$ 和 $\boldsymbol{\beta}$ 的值，可以计算出一个似然函数的值，称为 E-STEP。按最大化似然函数的目标，调整参数值，称为 M-STEP。反复执行这两步，直到收敛，就找出使得似然函数值最大的 $\boldsymbol{\alpha}$ 和 $\boldsymbol{\beta}$。这一过程中由于后验概率 $p(\theta,z|w)$ 没有解析表达式，需要一个函数来近似它。假设 θ 和 z 在给定 w 时条件独立，通过下界逼近的方法可以得到想要的解。该方法在[Blei 2003]中有详细介绍，这里不再作过多的解释。

关于 LDA 模型，由于近几年应用广泛，所以研究者甚众，网上也有很多相关的介绍，各种版本的实现方法在网上也能找得到（原作者的一个 c 实现在其主页上可以找得到：http://www.cs.princeton.edu/~blei/lda-c/index.html）。对于文本及其他媒体内容统计建模领域比较关注的读者，可以加深对 LDA 的了解，并作为理解其他模型的样例。

6.2.4　随机游走（Random Walk）

随机游走这一名称由 Karl Pearson 在 1905 年提出[Pearson, K. (1905). The problem of the Random Walk. Nature. 72, 294.]，本来是基于物理中“布朗运动”相关的微观粒子的运动形成的一个模型，后来这一模型作为数理金融中的重要的假设，指的是证券价格的时间序列将呈现随机状态，不会表现出某种可观测或统计的确定趋势，即证券价格的变动是不可预测的。在计算机领域，随机游走则主要用来进行一种关系的传递分析，如图 6-5 所示。

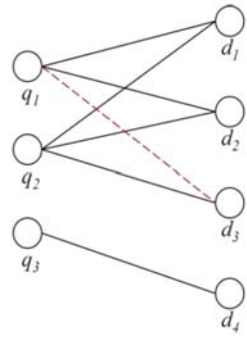


图 6-5　随机游走

以 q 表示查询，而 d 表示查询结果中的文档。则查询 $q1$ 所对应的文档为 $d1$ 和 $d2$，并不包含 $d3$。但不能简单断定 $q1$ 和 $d3$ 无关。因为查询 $q2$ 所对应的文档包含了与查询 $q1$ 相关的所有文档 $d1$、$d2$，此外还包含了 $d3$。文档 $d1$、$d2$ 和 $d3$ 通过共同的查询 $q2$ 建立了关联，而查询 $q1$ 和 $q2$ 则通过共同的文档 $d1$、$d2$ 建立了关联，关联性的传递意味着 $d3$ 也许是和 $q1$ 具有还未被表达的关联性。为表达这种关系的传递，可以视该图为一个随机场，依据结点间的连通性和转移概率进行随机游走，以传递结节的关联关系。

可以用一个简单例子来说明这种方法的实际意义，如某用户用 $q1$ 查询超市所售的糖，返回水果糖和奶糖两个结果;另一位用户则用 $q2$ 查询超市中所售甜食，不仅返回了水果糖、奶糖，还返回了巧克力。而巧克力实际上和第一位用户的查询意图相关性还是相当高的。

6.2.5　小结

对于搜索引擎日志分析中常用模型的介绍先进行到这里，在下面的内容里，大家将经常性地和它们打交道，在不同内容的日志分析中不断看到它们的身影，逐步加深对它们的理解。如果还想继续深入了解相关的理论知识，可以参考机器学习方面的课本，如[Bishop 2006]。有了这些铺垫，下文我们将按搜索过程更为详细地介绍搜索引擎的日志分析。

【注：大家应该了解的是日志分析并非始于搜索引擎，任何系统的使用过程记录都可以称为日志，日志分析的目标在于获取有价值的信息。从这个角度而言，日志分析可以视为是人工智能的一个应用领域。而这种从大量的真实数据中分析得到内在规律的方法，在人工智能领域内也是一种常用的方法，即大家所熟知的归纳式模

型。搜索引擎的日志分析也属性这一范畴，一系列的归纳模型均可以应用于从大量日志数据中分析得到有用的信息。】

6.3　查询日志分析

查询，是一次搜索行为的起始点，相应的，我们也从这里开始搜索引擎日志分析之旅。分析，首要的两件事，一是确定分析对象，二是确立分析目标。其中确定和理解分析对象更是确立分析目标的依据，所以我们要做的第一件事就是明确查询日志包含哪些内容。

6.3.1　查询日志的内容

举一个搜索日志的例子，来直观地感受一下查询日志都包含哪些基本元素。该例子来自[Jansen 2006]，是个比较典型的查询日志，麻雀虽小，五脏俱全。下面来看看它都具备了哪些查询日志的要素，如图 6-6 所示。

用户 ID	日期	时间	查询串
ce00160c04c4158087704275d69fbecd	25/Apr/2004	04:08:50	Sphagnum Moss Harvesting+New Jersey+Raking
38f04d74e651137587e9ba3f4f1af315	25/Apr/2004	04:08:50	emailanywhere
fabc953fe31996a0877732a1a970250a	25/Apr/2004	04:08:54	Tailpiece
5010dbbd750256bf4a2c3c77fb7f95c4	25/Apr/2004	04:08:54	1'personalities AND gender AND education'1
25/Apr/2004	**04:08:54**	**dmr panasonic**	
89bf2acc4b64e4570b89190f7694b301	25/Apr/2004	04:08:55	Bawdy poems
	“Mark Twain”	**25/Apr/2004**	
397e056655f01380cf181835dfc39426		**04:08:56**	**gay porn**
a9560248d1d8d7975ffc455fc921cdf6	25/Apr/2004	04:08:58	skin diagnostic
81347ea595323a15b18c08ba5167fbe3	25/Apr/2004	04:08:59	Pink Floyd CD label cover scans
3c5c399d3d7097d3d01aeea064305484	25/Apr/2004	04:09:00	freie stellen dangaard
9dafd20894b6d5f156846b56cd574f8d	25/Apr/2004	04:09:00	Moto.it
415154843dfe18f978ab6c63551f7c86	25/Apr/2004	04:09:00	Capability Maturity Model VS.
c03488704a64d981e263e3e8cf1211ef	25/Apr/2004	04:09:01	ana cleonides paulo fontoura

Note. 为保护隐私，粗体部分进行了处理。

图 6-6　搜索日志片段

首先大家会注意到这个数据集，包含的内容分 4 类，每类占据一列内容，依次是用户 ID、查询的日期、时间和查询串。此外，这个表中标加黑的地方表明这条日志的内容不全，某些信息没有记录完整，这个现象在工业搜索引擎中会有，但不会像这个例子中发生得这么频繁，而且这样非常明显缺信息项的日志在日志预处理的时候很容易被发现而剔除，所以这样的噪声信息在真正做日志分析的时候基本不会有影响。

查询日志，基本的要素就是 who、when、what，也就是谁在什么时刻查询了什么内容。结构简单明了，上面这个例子就完整地记录了这 3 项内容。当然日志中还可以包含用户的来源 IP 是什么，用的什么浏览器等其他信息等。但具有上述 3 个元素后，就是一个合格的查询日志了。

这么简单的查询日志，可以用它来做点啥呢？下面来看第一个应用。

6.3.2 查询词频统计

顾名思义，查询词频统计就是统计各个查询词出现的频率。这句话听起来有点无趣，不过要是加点料，再结合一下应用背景，就会变得有意思起来。还是继续举例子[Jiang 2010]，如图 6-7 所示。

Query String	Count
facebook	3,157 K
google	1,796 K
youtube	1,162 K
myspace	702 K
facebook com	665 K
yahoo	658 K
yahoo mail	486 K
yahoo com	486 K
ebay	486 K
facebook login	445 K

图 6-7 查询次数统计

这是统计的各个查询词的查询次数，并且按次数多少进行了排序，非常简单的词频统计。结合查询日志，如果统计的是某一天的词频，就能知道引擎的用户们今天对哪些内容最为关注。如果统计时间为 1 小时，那么就知道用户在这一小时内对

哪些内容最为关注。如果把一天 24 小时的统计结果按时间顺序进行整理，就能看出用户在不同时段最为关心哪方面的内容。按不同时间段长短，统计查询频率就能了解各时段搜索量最大的查询串，以及查询量上升最快的查询串。现在大家应该知道风云榜大概是怎么生成的了吧，其基石正是这些平凡无奇的查询日志，以及背后的千千万万搜索引擎用户。

这个只是词频统计比较直接的应用，词频统计与其他技术相结合还可以做很多其他的应用，如同音字的纠错等。查一个人名，例如“周杰伦”，但是输入的时候，查询词错输入为“周结伦”，或者“周结论”，返回结果少，而且质量比较差，这时可以启动一个纠错的流程。方法非常简单，列出输入串的拼音，然后在查询频率高的 TOP 词表里进行匹配，发现有匹配上的，则提示用户是否输入有误，并给出可能的 TOP 词。基于词频统计还可以根据词频进行一些统计分析，诸如高频词的查询次数占整个查询次数的比例，每天查询词的重复度等，对于有针对性地改进查询效果有着重要的作用。

由于查询频率高的查询串被用户使用的次数相对更多，所以通常对这样的串作统计时会比较关注。查询串可能是各种各样，并非是一个封闭集，如何实时统计这些高频的查询呢？可以这样来考虑：视查询串为一个数据流，每个查询串为一种模式，那么查询频率的统计就归结为在数据流程中统计各种模式出现的频率了。

比如要统计从某一天零时到某一时刻，频率最高的的 K 个查询，常用的做法是，对于接收到的查询串，进行 HASH，按 HASH 值把它放到某一个存储块内。可以通过分块数目的多少，来把各个块的大小控制在一个比较适合于进行内排序的规模上。对于接收到的查询串，不断地在相应块中记录它被查询的次数，隔一定的时间就对各个块按查询串的频率进行内排序，将各块的前 K 个高频查询取出，进行堆排序。堆构造好后，即可作为一个对于高频查询统计的 CACHE 来使用。在一个查询到来后，一方面通过 HASH，修改它在存储块内的记录，另一方面，可以通过在堆中查找它，直接更新它在堆中的频率记数，并对堆进行相应调整。通过这一套方法即可实现对于高频查询接近实时的统计。

有了查询词的频率统计信息，就可以开展一些其他的应用，一个比较典型的就是查询词的自动提示。

6.3.3　查询词提示（Suggestion）

查询词提示，现在基本已经是搜索引擎必备的一个功能。当你在搜索框进行输

入时，搜索框会打开下拉的提示框，动态地向你提示一些与你已经输入内容相关的查询串。如果在提示框中看到所希望输入的查询串，直接用鼠标点击或键盘选择后，即可进行搜索，减少你的输入字符数。说到这里，大家应该想到前面所说的高频查询统计是如何用在这当中的了吧。

一个基本的模型是：统计查询串中可能出现的各个前缀子串，然后统计以各个前缀子串开头的查询串的频率。对各个子串，取以其作为前缀的对应 K 个最高频查询串。再以各子串为 KEY，建立倒排索引。定期更新倒排索引，即可依据查询频率给出一个比较合理的提示内容。

模型并不复杂，但这么做有什么好处，其中包含的频率统计信息有什么意义呢？

来做个简单分析，比如一个搜索引擎，一天接收到的查询次数为 1000 万次，不同的查询串有 100 万个。通常而言，按查询频率来统计，频率较高的前 10%查询词，其被查询的次数会占到总查询次数的近 50%。下面用已经介绍过的二分图模型来说明这个过程，如图 6-8 所示。

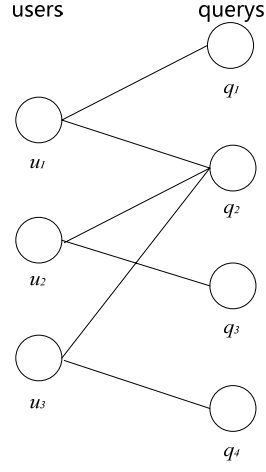


图 6-8　用户与查询的关联

对于用户和具有相同前缀的查询串，分别以上图中的 u 和 q 来表示。对于用户 u_i 查询 q_j 串的行为，以在 u_i 与 q_j 间连边来表示。假设各次查询之间是相互独立的，则一次查询命中查询串 q 的概率与 q 的度数成正比。如果将已有的查询记录作为先验模型，那么当某用户发起一次查询时，度数高，即查询频率高的串被命中的概率就大，所以会优先出以用户输入为前缀的高频查询串。

而具体到一个查询串，按上述思路来推测，当用户输入第一个字之后，所提示的字串中包含用户希望输入查询串的概率可以这样计算：提示字串在所有以该字开头的查询串中查询次数所占的比例。一个合理的假设是，以某串 P 作为前缀的查询串 Q，其数目会随着 P 的串长增长而快速下降。这个速度通过统计公共前缀查询串的数目可以推断，读者也能够查找到相关的统计资料，这里不再举具体数据。

按这样的方法，随着用户主动输入字串的长度增长，提示词命中用户真实查询词的概率将不断增加，因而需要用户主动输入的字串长度将缩短。整体上讲，能有效地降低用户使用搜索引擎的复杂度。这对于广大的中文搜索引擎用户，无疑是有吸引力的。因为大多数用户使用的是拼音输入法，需要不断地在同音字中选取自己所需要的字、词，现在搜索引擎有了这样一个功能，大家使用起来会方便很多。

这只是查询日志进行词频统计的应用之一。大家还记得这项功能是哪一年出现在搜索引擎中的吗？而在那个时期，这种思路对于中文输入法又产生了什么重大的影响？这些变化的背后，体现出了互联网的力量，千千万万互联网用户不再只是被动地使用互联网，他们还在通过互联网影响搜索引擎、输入法以及其他信息系统的变革，而这一力量正是直接源于用户日志。

6.3.4　命名实体（Named Entity）类别识别

除了在预测用户意图方面的用途，查询日志还可以用来识别命名实体。命名实体识别是指识别文本中具有特定意义的实体，主要包括人名、地名、机构名、时间、日期、货币及其他专有名词等。它是自然语言处理实用化的重要内容，在信息提取、句法分析、机器翻译等应用领域中具有重要的基础性作用。命名实体识别一方面要识别实体边界，另一方面要识别实体类别（人名、地名、机构名或其他）。就汉语系统来讲，确定实体边界主要和分词相关，发现命名实体的基本方法，一般首先找一些与定义相关的特征词，例如：什么是 XX，XX 是什么，这是 XX。找到具有这样模式的查询串后，即可以在查询日志中通过频率统计等方法，找到命名实体。这里重点讨论第二方面的内容，即类别识别。

之所以会用查询日志来进行命名实体的类别识别，是因为命名实体的类别并非是一个封闭集，而是一个不断变化着的集合。一个命名实体，随着时间的变化，往往会具有不同的属性。以大家熟悉的“哈利·波特”为例，它开始是一部小说，然后又推出了同名的电影，后来还出了游戏，而这一过程是随着时间变化的，也就是说在不同时间段，这些类别在用户查询需求中受关注程度是不一样的。

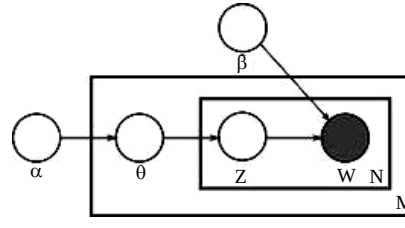
类别分析首先统计和命名实体相关的查询，如图 6-9 所示[Jiang 2010]。

查询词	次数
final fantasy	300
final fantasy movie	120
final fantasy wallpaper	50
gone with the wind movie	120
gone with the wind review	10
gone with the wind photos	10
harry potter	1000
harry potter book	650
gone with the wind book	80
gone with the wind summary	20
harry potter movie	800
harry potter pics	200
final fantasy xbox	10

[注：这里只摘录了部分数据集]

图 6-9　命名实体的查询次数

对于这些查询，如果要把它对应到相应的 3 种类型：书、游戏和电影上去，可以采用 WS-LDA(Weakly Supervised Latent Dirichlet Allocation)模型[Guo 2009b]，通过训练的方式得到对于某个含实体的查询词属于某一类型的概率模型，然后在查询端采用训练出的模型判定某个含实体查询潜在的类别，如图 6-10 所示。



z: *Movie*, *Book*, *Game*

w: \#, \# *movie*, \# *book*, ….

θ: distribution of classes for named entity

β: distribution of contexts for class

图 6-10　实体分类 LDA 模型

以电影、图片、小说、摘要、评论等与实体相伴随的词和事先标注的样本集作为训练对象，得到以下的概率分布，如图 6-11 所示。

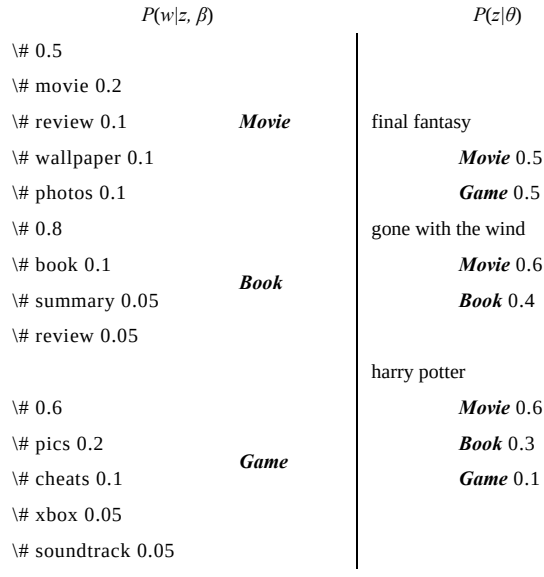


图 6-11　实体分类训练集

当实际查询某个实体时，根据含该实体的查询中伴随次出现的频率，作为 w，即可用上述模型进行预测，以得到某查询属于电影、书或是游戏的概率。

6.3.5　小结

关于查询日志的分析和应用，先介绍到这里。查询日志记录的还只是用户发给搜索引擎系统的请求，这种请求还仅仅是单向的，它就已经具备了这样的力量。下面我们将按搜索的流程继续讨论下一个阶段的日志——点击日志，这类日志中既包含了搜索引擎对用户的应答信息，又包含了用户对系统应答的反应。在这一阶段，人和机器的交流已经是双向的，其中的含义将更为丰富，值得讨论的内容也将更多。

6.4　点击日志分析

点击日志，一方面包含了搜索引擎返回的结果信息，另一方面还包含了用户对

这些结果的反馈行为。一来一去，就包含了用户与系统二者之间的交互信息。较之查询日志的单向性，可利用的信息更多，应用范围也更广。我们先从介绍点击日志所包含的内容开始吧。

6.4.1 点击日志的内容

点击日志，广义而言，记录的是用户对搜索结果的反馈，包含用户的 Click-through（即点击搜索结果，到最终页面去）行为、浏览行为、驻留时间或是点击相关推荐列表的行为等。由于这些行为一般都在客户端（浏览器）发生，所以经常通过浏览器端记录，并向服务器传递信息而得到。

还是先看一个例子[Guo 2009a]，如图 6-12 所示，它记录了在 bing 中某一次查询后的点击情况。查询词为 cikm，在前 10 个结果中第 1 个结果和第 5 个结果被点击。

Query	cikm	Session ID	f851c5af178384d12f3d
Position	URL		Click
1	cikm2008.org		1
2	www.cikm.org		0
3	www.cikm.org/2002		0
4	www.fc.ul.pt/cikm2007		0
5	www.comp.polyu.edu.hk/conference/cikm2009		1
6	cikmconference.org		0
7	Ir.iit.edu/cikm2004		0
8	www.informatik.uni-trier.de/~ley/db/conf/cikm/index.html		0
9	www.tzi.de/CIKM2005		0
10	www.cikm.com		0

图 6-12 点击日志例子

图 6-13 进而统计了对这一查询词，多个用户的点击情况。

这些就是点击日志所记录的基本内容了，此外，各个点击发生的时间点也会作相应记录。这样，用户浏览到了哪些结果，点击了哪些结果，还有先点击的是哪个，停留时间有多长等，都可以从点击日志中找到相应的信息。

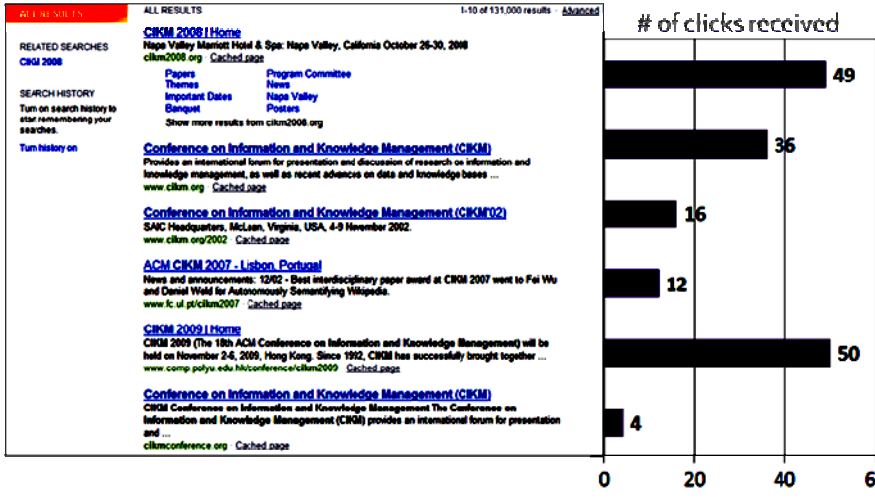


图 6-13　多用户点击日志统计

前面已经说到，有了点击日志就有了用户与系统两者之间的交互信息。这一次交互将带来什么有利因素呢？下个小节以前面已经处理过的查询串提示问题为例，看看在其中加上点击交互信息，会有什么帮助。

6.4.2　查询串提示（Suggestion）再分析

上节已经介绍过了使用查询日志进行的 Suggestion 方法，在引入了点击日志之后，还可以进一步利用点击信息来度量查询间的相似性，从而改进 Suggestion[Cao 2008]。

以二分图作为分析工具，对于查询 q_i，用户点击的结果 URL 为 u_j，则在与之间连一加权边，e_{ij} 其权重为对于 q_i 的所有查询中，u_j 被点击的次数 w_{ij}。如图 6-14 所示。

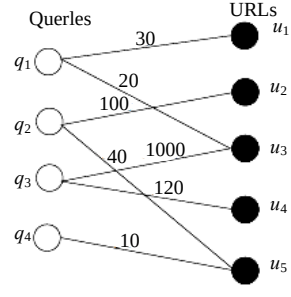


图 6-14　查询与点击结果的关联

对于查询 q_i 则可以用这些与之相连的加权边表出。其形式为：

$$\vec{q}_i[j] = \begin{cases} \dfrac{w_{if}}{\sqrt{\sum_{\forall e_{ik}} w_{ik}^2}} & \text{if } e_{ik} \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

基于这种表示，就可以度量两个查询之间的相似度：

$$distance(q_i, q_j) = \sqrt{\sum_{u_k \in U} (\vec{q}_i[k] - \vec{q}_j[k])^2}$$

这种做法就是利用点击作为联系查询与结果的桥梁，将结果集作为表达查询的一个特征空间，并在这个特征空间中度量查询的相似度，从而找到相近的查询作为 Suggestion 结果，由此点击日志的作用可见一斑。

6.4.3 查询和结果类别属性传递

利用点击信息在查询端不仅可以做 Suggestion，还有助于进行关键词提取和内容分类。[Fuxman 2008]中同样是采用上述加权二分图模型作为出发点，依然是在查询 q_i 与用户点击的结果 u_j 之间连加权边。不同的是，通过首先对部分结果进行类别标识，进而利用边的传导性，将这种类别属性向未标识的结果上进行传导。一方面可以获得未标识类别结果的类别属性，另一方面也可以将和同一类结果相关联的查询进行聚类。

这一方法的主要思想在于利用群体行为来进行属性传递。回顾前文所介绍的随机游走模型，如图 6-15 所示。

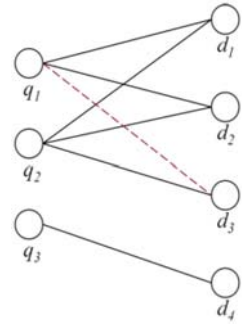


图 6-15　属性传递的随机游走模型

在这个应用环境下，先将其视为二分图模型，以点击次数为权重，对查询 q 与结果 u 间的边进行加权。视这个加权图为一个随机场，依权重为转移概率进行随机游走，从而将 q 或 u 结点上已经标注的属性逐步扩散到未标注属性的节点上。

这一点在广告系统中很有用。试想，如果知道某些商业网站是和某类购物需求相关的，那么通过上述传递方法，可以找到与这些网站类似，而尚未被标注类别属性的网站。同时，还可以发现一些与这类购物需求相关的查询，以达到准确投放相关广告的目的。

6.4.4　搜索结果相似性度量

前文讲过用结果作为特征集来对查询进行表达的方法，反之，也可以用查询作为特征集，来表达搜索结果[Xue 2004]，如图 6-16 所示。

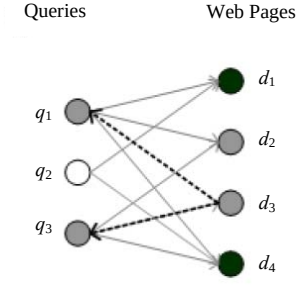


图 6-16　结果相似性与查询的关联

上图是一个有向二分图，实线箭头表示查询 q 对返回结果 d 的点击。可以用下面的公式来计算结果间的相似度：

$$S(d_i,d_j)=\frac{visited(d_i,d_j)}{visited(d_i)+visted(d_j)-visited(d_i,d_j)}$$

以 d_2，d_3 为例，即点击了 d_2 又点击了 d_3 的查询仅有 q_3，visited(d_2，d_3)=1;而点击了 d_2 的查询则有 q_1 和 q_3，visited(d_2)=2；点击 d_3 的查询仅有 q_3, visited(d_3)=1。因此 S(d_2，d_3)=1/(2+1-1)。这个方法的本质就是：计算点击了某结果集中全部结果的查询个数与点击该结果集中任意结果的查询个数之比，以衡量该结果集中结果的相似性。

反之，也可以用类似的方法来度量查询词的相似性，模型相仿，而相似度计算方法可以定义为查询集中查询间的相似度，即用该集中所有查询都点击的结果数来

比这些查询所点击的所有结果数。

进一步地，考虑到同一查询会反复多次进行，将二部图的有向边按点击次数进行加权，可以得到更复杂的模型。这里只介绍基本的模型，加权模型大家可以参考[Xue 04]。

6.4.5 查询结果排序

点击日志的另一个重要用途就是对结果进行排序。用户给出查询需求，在看到查询结果后，一般而言，只会点击感兴趣的搜索结果。如果点击的位置不在给出的前排结果中，可以认为排在前面的结果并没有很好地满足用户的需求。因此，用户的点击是对搜索结果的一个反馈，也是对结果排序是否合理的暗示。

通常而言，被点击的结果应该比未被点击的结果更符合用户的需求[Joachims 2002]，所以在排序上应该更靠前。进一步地，还需要考虑被点击结果之前的优先程度，以及未被点击结果间的优先程度。回到本节开始的 cikm 查询例子，对于被点击的第 1 个和第 5 个结果而言，谁更优呢？先来看一个统计图[Guo 2009a]，如图 6-17 所示。

图 6-17　页面中用户关注位置分布

该图表示的是结果页不同部位受用户关注的程度差异，深色部分表示受到更高

的关注。由此导致的是，用户的点击行为是一个有偏见的行为，排在前端的结果被点击的概率会更高。即使将原来的排序倒过来，将算法得到的相关性较差的结果放在最前，也仍然有这样的趋势，统计点击日志就能得到相关的信息。图 6-18 就是一个这样的统计结果[Joachims 2007]。

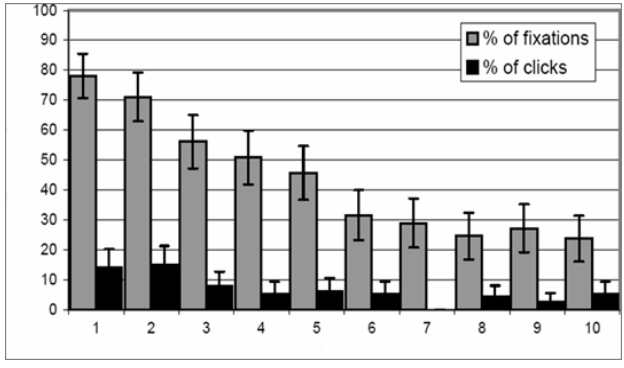


图 6-18　关注位置与点击位置对比

图中统计了不同位置受目光注视的比例，以及第一次点击发生在该位置结果的比例。这样的结果显示，排序相当重要，对用户体验有着非常大的影响。从 cikm 例子来看，第 5 个结果受到关注的概率较小，但它被点击的次数却不少，说明它是个不错的结果，可以往前排。关于点击所反映出的排序相关信息[Joachims 2005]作了形式化的分析，其基本观点是：对于排序结果$(d_1, \ldots, d_n)$而言，如果结果 d_i 被点击，那么它优于排在它前面而没有被点击的结果。

如何利用点击日志进行排序优化呢？直观的想法是，给出一个排序结果，预测它可能的点击情况，并使得期望的点击情况出现。但是有个实际问题，还是以 cikm 查询为例来说明，对该查询首页的 10 个结果，每个结果有可能被点击或者不被点击，它们总的点击可能情况是 2^{10}，在这样大的状态空间来预测某个排序的效果，显然不是一个好方法。此外，依靠两结果之间的优先次序来进行整体结果的排序也会有实际困难。这是由于在不同的查询中，会出现针对某个查询，结果 *a* 优于结果 *b*，结果 *b* 优于结果 *c*，而在另一个查询中 *c* 却优于 *a* 的状况。结果间的相互优先关系是不稳定的，因而针对各个查询专门建立优先关系模型也不现实。

在实际应用中，排序主要还是依靠考察结果与查询意图间的匹配性来解决。对于一次查询，给每个搜索结果打一个相关性分数，让得分高的排在前面，这就是排序问题的基本思路。这类方法也利于搜索引擎系统实现。不论是有点击信息还是没有点击信息的数据，对于一个查询都可以有一个分数来度量，供最终排序参考。而

如何在这一体系中引入点击信息，优化排序结果呢？这里以基于用户点击行为推测用户偏好，并根据偏好进行重排序的方法[Zhao 2006]为例，介绍一类基本模型。整体上，该模型首先依据查询和结果间的文本相关性得到一个排序结果，然后再进行修正，如图 6-19 所示。

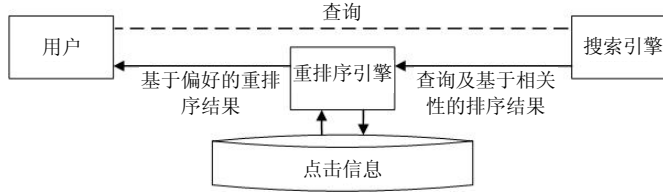


图 6-19　重排序的基本框架

在进行修正的过程中，运用点击日志的方法可以用基于查询—结果点击二分图的方法来描述，如图 6-20 所示。

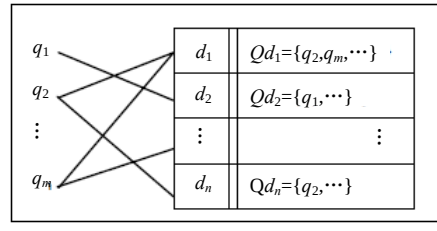


图 6-20　查询串与结果文档的关联二分图

对于查询结果 d，考查点击了该结果的查询集 Qd，统计 Qd 中词的词频 tf。即针对某个特定的查询 q，可以依据 q 中所包含的属于 Qd 的词，通过词频来计算 q 和结果 d 之间的关联性。实质上就是用一个由点击行为所生成的，与 d 相关的词频信息，来估计某个文档 q 与 d 的相关性。

目前搜索引擎用点击信息来进行排序优化是一个很活跃的领域，是 Learning to Rank 的一项重要研究内容。由于本书将专门有章节介绍 Learning to Rank，大家会看到点击信息更丰富、更精彩的应用，此处不再赘述，而是引入另一项也非常有趣的内容。

6.4.6　点击数据的稀疏性

搜索引擎收录的数据量相当大，但出于响应速度等考虑，用户搜索后所展现的结果只是其中的很小一部分。在这些展现的结果中，一般也只有排序靠前的极少一

部分被用户点击的次数较多。此外，查询词的频率分布差异也很大，相当多的查询词被用户查询的次数并不多，而少量查询词则多次被用户查询，这一点在查询日志分析中已经提到过。这会导致很多查询词，相关的用户点击反馈量很少。

对于由查询频率低导致的用户点击少的问题，解决的基本思路是通过对点击结果进行聚类，以类别为单位进行点击数据的分析[Yates 2004]。首先，通过点击日志将查询和点击的结果组成数据对；而后对被点击的结果根据文本内容进行聚类，对于聚类结果 C，给出类别 C_i 中结果相对应的查询集 Q_i，并统计这一类结果中点击频率较高的一部分结果，作为这类查询集中应优先给出的结果集。

对查询 q，首先确定它的类别归属，然后对于返回的结果，融合相应类中的优先结果集信息，给出最终结果。利用结果的聚类性，将点击关系从单一的查询、结果对扩展到了查询——结果类上，从而在不存在点击映射关系的查询与结果之间建立了联系。这在一定程度上缓解了某些查询频率低造成的影响。

另一方面，对于大量数据只有很少甚至没有点击信息的问题，则可以采用前文介绍过的随机游走模型进行处理。其基本思想是通过随机游走，将少数的查询——结果关联信息逐步地扩散开，以得到覆盖面更广的关联性[Gao 2009]。以 q_i 表示查询，而以 d_i 表示结果文档，二者及其之间的点击关系可用二分图模型表示，如图 6-21 所示。

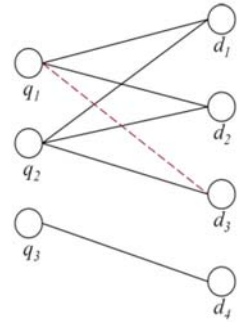


图 6-21　查询与结果关联性的扩散

图中当用户查询 q_i，以点击结果 d_i 时，在两者之间连接一条边，以表达点击关系（可以用点击次数对边进行加权）。以 A_{ij} 记查询 q_i 时点击 d_i 的概率，$A_{ij}=P(d_i|q_i)$，则以 A_{ij} 为元的矩阵可以作为在查询端进行随机游走的转移概率。同样，可以构造矩阵 $\boldsymbol{B}$，$\boldsymbol{B}_{ij}=P(q_i|d_i)$作为在结果端的转移概率。

简单地讲，假设上图中每条边表示一次点击，那么从初始状态开始，结果 d_3 仅为查询 q_2 点击，所以 $P(q_2|d_3)$为 1，而 q_2 有三个可能点击的结果 d_3、d_3、d_3，所以 $P(d_3|\ q_2)$为 1/3。而 $q_1|d_3$ 间没有连边，所以 $P(q_1|d_3)$和 $P(d_3|q_1)$都为 0。同时有 $P(d_1|q_1)$=1/2，$P(d_2|q_1)$=1/2，$P(d_1|q_2)$=1/3，$P(d_2|q_2)$=1/3，$P(d_3|q_2)$=1/3，$P(q_2|d_1)$=1/2，$P(q_2|d_2)$=1/2。起始时，q_1 与 d_1 与 d_2 的关联性都是 1/2，然后通过概率转移，这种关联性经 d_1 与 d_2 传递到 q_2，其大小为 1/2* $P(q_2|d_1)$+ 1/2* $P(q_2|d_2)$=1/2，再经 q_2 传递到 d_3，其大小为 1/2* $P(d_3|q_2)$= 1/6。到这里，一共进行了三次转移，q_1 与 d_3 有了关联。这样的转移可以反复进行，直到收敛或者达到所期望的程度，以此达到扩展关联关系的目的。

但这一方法的局限性在于，能传递到的结节，必须相互之间是有路径可达的，上图中 q_3 与 d_4 就无法和其他结点发生关联。进而言之，在实际搜索中，没有点击信息的结果，就更不能和其他结点发生关联了。上述方法还有继续改进的空间，因此[Gao 2009]还提出了类似于 Good-Turing（图灵估计）的方法，通过对没有点击信息的数据进行平滑来解决稀疏问题。Good-Turing 在统计语言模型常用来平滑不常出现的 N 连文法 (n-gram) 。对于出现 r 次的 n-gram 来说，经过估计后，新的出现次数 $r*$ 为：

$$r^* = (r+1)\frac{n_{r+1}}{n_r}$$

其中 n_r 代表 n-gram 在训练集中出现 r 次的个数。

但这一模型不能直接用于查询和结果的点击关联数据上，对相互之间不存在联系的数据对添加联系。这是因为查询和结果的相关性不仅和点击数有关，还和其他因素如文本匹配度等因素有关。如果直接对点击数进行平滑，因为有无穷多种查询串的可能形式，得到的查询和结果相关性，将直接被点击信息所左右。为此，该文所提出的方法用查询作为特征，对与之相关联的结果进行表出。其中既包含了关联查询的个数，也包含了查询串内容。而后直接对结果的表出特征进行基于关联数平滑。具体的做法有兴趣的读者可以参考[Gao 2009]。

6.4.7 小结

点击日志包含了用户与系统双向交流的信息，含义丰富，值得讨论的内容不胜枚举，限于篇幅和作者的眼界，本节所介绍的内容只能说是挂一漏万。这个领域的研究和应用目前均非常活跃，新颖有趣的思路和方法层出不穷，与其他领域的交叉

研究也不少。有兴趣的读者按本节所介绍的几方面内容逐步进行扩展，可以发现更多值得关注的内容。

6.5　隐私问题

隐私者，一般是不愿告人或不便告人的事情。那么大家认为在互联网上，什么情况下会比较容易泄漏自己的隐私呢？在进行用户注册时？被木马攻击时？没错，这些时候的确是容易泄漏隐私。但大家有没有想过，使用搜索引擎也会泄漏隐私呢？又有没有想过在搜索日志里可能包含着人们不愿别人知道的隐私呢？这一节，就要探讨这个大家不愿意提及，但又不得不面对的问题：搜索引擎日志的隐私问题。先从搜索日志的可用性说起吧。

6.5.1　日志的两面性

搜索引擎是一项伟大的发明，但它的功能只有在明确用户需求的条件下才能达到。它只有知道了用户需要什么，才能帮助用户找到所要的东西（完全理解用户的查询意图，目前搜索引擎可能还达不到，但这的确是搜索系统改进的初衷和终极目标）。广而言之，任何服务系统都必须首先明确用户需求才能提供令用户满意的服务。只是搜索系统所拥有的信息量规模太大，所以大家基本都快把它当成阿拉丁神灯一样的有求必应系统。

本章前文介绍了日志分析的各种方法和相关用途，这些用途和这些方法都是基于搜索日志中人的因素。用户的参与使得搜索日志具备了独特的可用性，而这种可用性从根本上来讲，正是由于用户在与系统的交换中表达了其意图。一方面，查询词是用户意图的一种直接表示，而不断地点击、浏览搜索结果，并修正查询词，则是用户意图确认、修正，逐步清晰化的过程。

日志的可用性来源于用户的意图表达和理解。但同时，用户意图的显露，也会反映出用户某方面的属性。举例而言，我们有去的小吃店或快餐厅，如果每次去，某人都点甜豆浆，那么服务员就会意识到，这是一位喜欢吃甜豆浆的顾客，甚至进一步推断这是位喜欢甜食的顾客。继续举例，这位顾客到服装店买衣服，如果他选择的都是宽松型号的服饰，那么他很可能有点胖。按这样的模式，如果汇集一个人各方面的需求，那么就很有可能了解这个人的方方面面。但由于人们平时接触的服

务系统一般都只是满足某一方面的需求，所以任何一个单一系统都不太容易完整地刻画和确认某个人物。

不过搜索引擎就不太一样了，它是一种用户与信息世界交互的基本介质系统。一个人在信息世界的各种需求都有可能通过它表达出来，所以通过广泛了解用户的搜索需求和搜索行为，就可能得到更为全面的用户信息。回到前面吃甜食和买衣服的例子，如果这位用户通过搜索网店，或是货品的方法，找自己喜爱的豆浆和服装，进而在网上购买，分析日志就能推测这是一位有点胖但是又喜欢甜食的人。

依据这一点，可以给这位用户投放木糖醇系列的食品广告，并通过分析广告的点击情况来进行进一步的改进。这对于服务提供者（食品厂商、服务厂商、木糖醇供应厂商和搜索引擎）以及用户而言，都是件不错的事。一个可以获取有效消费，另一个则能得到方便甚至适合自己潜在需求的服务。用户通过搜索引擎获得的服务远远不只这个例子这么简单，各种各样的需求都可以在搜索引擎系统中得到不同程度的满足，同时也会更多地显露自己的相关信息。喜欢吃什么，什么时候睡觉，甚至家里用什么型号的冰箱等，诸如此类，不胜枚举。

对搜索引擎而言中，更多地了解某个用户，可以提供更好的服务。试想，如果搜索引擎彻底地了解了用户的需求，那么每次查询，第一个结果就是用户想要的，而且还能向用户推荐进一步想通过搜索了解的内容，这是种什么感觉？这基本是搜索的终极目标了，但享受如此高质量的服务也是需要付出代价的。

用户使用搜索引擎越频繁，在系统中表达自己的需求越多，那么搜索引擎就会更全面、准确地了解用户。但同时，用户个人方方面面的性质就会暴露得越多。事物并不是总有美好的一面，这种个人信息的暴露有时候会带来不良的后果。一个著名的反面事件发生在 2007 年。AOL（America Online）发布了一个时间跨度 3 个月，涉及 65 万用户的搜索日志。AOL 在日志中，将用户 ID 用随机数代替，试图掩盖用户的真实身份和隐私。但这件事演变出近乎灾难的后果，在《纽约时报》的一篇报道[Barbaro 2006]中，一位生活在现实世界中的活生生的人，通过这些日志被定位，其本人和朋友在日志中体现的隐私被曝光。随后，AOL 的 CTO 因这一事件辞职，而且还引发了一场针对 AOL 的旷日持久的诉讼。

这样的事件给当事人造成了巨大的伤害和重大的损失，但搜索日志中所蕴涵的丰富信息，无论对搜索引擎厂商、其他服务厂商、相关研究者，还是对用户都有如此重大的意义，以至于人们做出种种尝试，力图在保护用户隐私的同时还能有效地利用搜索日志。

日志就是这样具有两面性，利用好了能带来利益和便捷，处理不当则会带来灾难。

6.5.2　日志的安全使用

要保护用户的隐私，又要对日志进行分析和利用，这是一对矛盾，保护性地进行利用搜索日志这一研究领域也因此而兴起。需要保护用户的隐私信息通常有身份证号、信用卡号、手机号、地址等和个人身份直接相关的敏感信息，还有涉及年龄、性别、健康状况、经济状况等一些间接相关的信息。

讨论如何保护这些相关的隐私信息，需要从这些信息可能暴露的方式开始分析。通常，要分析某个 ID 的信息，首先是将与之相关的各个查询和这个 ID 相关联，而后从这些关联的查询中分析得到隐私信息。

针对这种基本模式，保护隐私信息的首选手段就是切断查询与用户身份的关联。一般方法是去除查询中所有暴露用户身份的信息。这样，即使同一 ID 的查询都被关联到一起，也无法得到这一组信息和真实用户身份的关联。除此以外，删除日志中某些具有明显特征的内容也是一种选择，比如删如数字串，那么身份证、手机号等一类信息基本上就能被清除。需要注意的是，如果只是简单通过 HASH 的方法来隐藏某些关键信息，当攻击者同时具有大量处理与未处理的日志时，可以通过词频分析等方法对隐藏的内容进行有效猜测。

总体上，常用的方法一般可分为以下几类：直接删除敏感日志；对日志中的查询内容进行 HASH；删除用户 ID 信息；对用户 ID 进行 HASH；清洗个人相关信息；缩短日志分析的时间跨度，不长期保留同一用户 ID 的相关信息；去除低频率的查询日志，以降低日志信息量，降低暴露隐私的可能性等。

目前，对于各种处理方法的隐私保护强度、日志可用性保留程度都还缺乏统一的衡量标准。已经提出的衡量标准有 k-anonymity（k 匿名：即一个人的信息和至少另 k-1 的信息无法区分）等。这些方法涉及不少传统上属于密码学领域的知识和理念，本章不作展开。如果希望深入了解，读者可以参考[Cooper 2008]和[Götz 2010]这两篇综述性的论文。

6.5.3　小结

日志的保护隐私性要求，使得公布搜索日志是一件具有很大风险的行为。但为

了推动日志分析研究更快地发展，公布日志数据以便研究、分析、利用，也是一件非常有意义的事。所以尽管隐私保护领域在日志分析中还起步不久，但其受重视程度必将随着搜索日志的更广泛应用而日渐增加。

6.6 本章总结

本章介绍了搜索日志分析中常用的概念和模型，并依照搜索流程介绍了查询和点击阶段的基本日志分析方法，所介绍的内容和方法还需要读者结合搜索日志分析的实际应用来掌握和发展。在掌握了这些基本内容之后，读者还可以涉足基于日志的搜索个性化、用户行为建模等内容，甚至在实际应用中找到新的切入点。此外掌握相关工具更可以达到事半功倍的效果，比如 Hadoop 和 MapRecue 架构等。本章的主题所限，这里不再做进一步介绍，读者可以参阅相关主题的文献和技术书籍。总之，在这个富有朝气的领域，不断地有新的思路和方法涌现，本章的介绍只是一个引导，对这个领域的理解和领悟还需要大家不断地思考和探索。

参考文献

[Barbaro 2006] Michael Barbaro and Tom Zeller. A face is exposed for aol searcher No. 4417749. New York Times http://www.nytimes.com/2006/08/09/technology/09aol.html?ex=1312776000en=f6f61949c6da4d38ei=5090.

[Bishop 2006] Pattern Recognition And Machine Learning. C.M.Bishop. Springer, 2006.

[Blei 2003] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. J. Mach. Learn. Res., 3:993 – 1022, 2003.

[Cao 2008] H. Cao, D. Jiang, J. Pei, Q. He, Z. Liao, E. Chen, H. Li, Context-aware query suggestion by mining click-through and session data, the 14th ACM SIGKDD (:2008).

[Cao 2010] Bin Cao, Dou Shen, Kusansan Wang and Qiang Yang. Clickthrough Log

Analysis by Collaborate Ranking, the 24th AAAI, Atlanta, Georgia, USA. July 11-15, 2010.

[Cooper 2008] Cooper, A. A survey of query log privacy – enhancing techniques from a policy perspective. ACM Transactions on the Web. 2008.

[Fuxman 2008] Fuxman, A., et al. Using the wisdom of the crowds for keyword generation. WWW'08.

[Gao 2009] Gao, J., et al. Smoothing clickthrough data for web search ranking. SIGIR'09.

[Götz 2010] M Götz, A Machanavajjhala, G Wang, X Xiao, J Gehrke, Publishing Search Logs–A Comparative Study of Privacy Guarantees, In TKDE Transactions on Knowledge and Data Engineering, 2010.

[Guo 2009a] Fan Guo and Chao Liu. Statistical models for web search clicks log analysis. *CIKM '09*, Hong Kong, China, November 2009.

[Guo 2009b] Jiafeng Guo, Gu Xu, Xueqi Cheng, and Hang Li, Named entity recognition in query, Proceedings of the 32nd ACM SIGIR conference, 2009.

[Jansen 2006] Bernard J. Jansen, Search log analysis: What it is, what's been done, how to do it, Library & Information Science Research, 28 (2006).

[Jiang 2010] Web Search/Browse Log Mining: Challenges, Methods, and Applications, WWW 2010 Tutorial, April 26–30, 2010, Raleigh, North Carolina, USA.

[Joachims 2002]　T. Joachims. Optimizing search engines using clickthrough data. KDD’02.

[Joachims 2005] T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay. Accurately interpreting clickthrough data as implicit feedback. SIGIR'05.

[Joachims 2007] T. Joachims, L. Granka, B. Pan, H. Hembrooke, F. Radlinski, and G. Gay. Evaluating the accuracy of implicit feedback from clicks and query reformulations in web search, ACM TOIS, 25(2), 2007.

[Winn 2004] Variational Message Passing and its Applications. John M. Winn. Ph.D.

dissertation, 2004.

[Xue 2004] G.-R. Xue, H.-J. Zeng, Z. Chen, Y. Yu, W.-Y. Ma, W. Xi, and W. Fan. Optimizing web search using web click-through data. CIKM '04.

[Yates 2004] Ricardo Baeza-Yates, Carlos Hurtado, and Marcelo Mendoza. Query Clustering for Boosting Web Page Ranking AWIC, 2004.

[Zhao 2006] Min Zhao, Hang Li, Adwait Ratnaparkhi, Hsiao-Wuen Hon, and Jue Wang. Adapting document ranking to users' preferences using click-through data, AIRS’06.

第 7 章　排序学习（Learning to Rank）

7.1　排序概述

7.2　传统的排序模型

7.3　排序学习简介以及研究现状

7.4　排序学习模型的应用实例

7.5　排序学习方法的框架

7.6　评测数据集

7.7　排序学习模型简介

7.8　排序学习模型性能比较

7.9　排序学习的研究方向

7.10　总结

7.1 排序概述

排序是众多信息检索应用中一个核心的问题[Qin 2010]，如文档检索[Cao 2006]、协同过滤[Harrington 2003]、关键词提取[Collins 2002]、命名实体识别[Xu 2005]、电子邮件路由[Chirita 2005]、情感分析[Pang 2005]、产品评价[Dave 2003]、反垃圾[Gyongyi 2004]等。在这些应用中，针对待排序的对象（如文档、网页、邮件等），利用排序模型（或者称为排序函数，这两个在本章中表示的意义相同）计算每个对象的分数，并根据分数给出所有对象的排序。根据应用的不同，这个分数可以用来表示相关、倾向或者重要程度。

本章中提到的排序，主要应用于文档检索，其中文档的形式有很多种，如网页、邮件、学术论文、书籍等，而搜索引擎是文档检索的一种应用形式。

中国互联网络信息中心（CNNIC）于 2010 年初公布了近几年关于网页数量与增长率的统计结果，如图 7-1 所示。

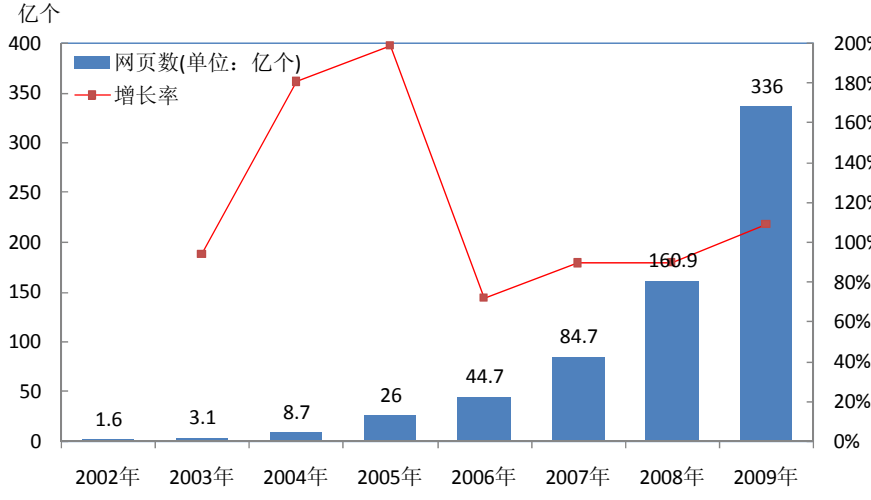


图 7-1　中国互联网络信息中心于 2010 年 1 月公布的近几年网页数量与增长率的统计结果

从图中可以看出网页数量正以指数级增长，已经达到一个相当大的规模。互联网络充斥着各种信息，当用户需要寻找某一方面信息的时候，往往求助于搜索引擎。

搜索引擎的基本结构可以用图 7-2 表示。当用户输入查询词“走进搜索引擎”时，搜索引擎通过排序模型对文档索引库中的文档进行排序，并将文档排序结果返回给用户，从中可以看出排序模型是搜索引擎中一个重要的环节，它的目标是在返回给用户的文档序列中，将满足用户需求的网页尽量排在文档序列的前面。

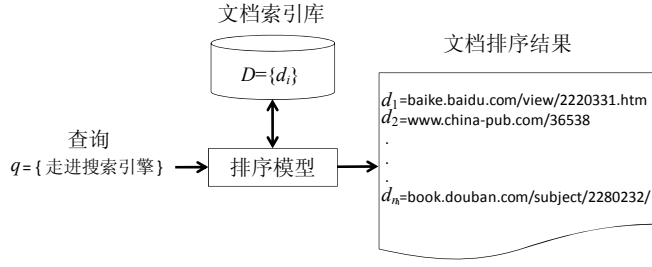


图 7-2　搜索引擎基本结构图

传统的排序模型可以划分为两个发展阶段，第一个阶段是基于词频和位置统计的排序模型，如布尔模型、向量空间模型、BM25 模型、语言模型等；第二阶段是基于链接分析的排序模型，如 PageRank 模型、HITS 模型等。

如何构建排序效果更优的排序模型一直是企业界和学术界关心的问题，传统的排序模型在解决这一需求时，遇到了很多的问题。

（1）对于模型参数的调整，当模型需要调整的参数数量很大的时候，传统的排序模型不能很好地处理这一问题。现在影响文档排序的因素，即文档排序特征，在实验用的评测集数据集（参见 7.6 节）上，就有数百种之多，基于内容的特征，如 TFIDF、BM25；基于链接的特征，如 PageRank、HITS；基于点击数据的用户行为特征等。

（2）对于模型的整合，如何将若干种排序模型有效地整合起来。如上面提到的排序特征 BM25 和 PageRank，它们本身就是一个排序模型，如何将它们整合成为更优的排序模型。

（3）对于两个排序模型性能的比较，两个排序模型对一个测试集中的文档进行排序，如果其中一个排序模型是过拟合的，那么如何比较两个排序模型的性能。

机器学习被证明是一种非常有效的工具，对上述问题可以给出很好的解决办法，因为机器学习可以做到：（1）自动对模型中的参数进行调整。（2）融合多个模型的结果。（3）通过结构风险最小化（Structure Risk Minimization）等方式避免过拟合。

因此，一个新的研究领域，排序学习（Learning to Rank）应运而生，它是信息检索与机器学习的交叉学科，并越来越受到人们的重视。

7.2 传统的排序模型

按照排序模型与查询之间的关系，传统的排序模型可以分为两类：查询相关的排序模型和查询无关的排序模型。本节分为两部分，分别以具体的排序模型为例子，介绍查询相关的排序模型和查询无关的排序模型。

7.2.1 查询相关的排序模型

查询相关的排序模型，是指相对一个查询，按照与查询的相关程度，对文档进行排序，已经提出的模型有布尔模型（Boolean Model）、向量空间模型（Vector Space Model）、Okapi BM25 模型、语言模型（Language Model）等。下面以向量空间模型和 Okapi BM25 模型为例，简单介绍查询相关的排序模型。

向量空间模型

Salton 等提出了向量空间模型[Salton 1975]，它是利用词的权重向量的方式来表示查询和文档，如查询 q_i 以及它对应的一篇文档 d_i^j，它们可以分别表示为：

$$q_i = (w_{1,i}, w_{2,i}, \cdots, w_{n,i})$$

$$d_i^j = (w_{1,i,j}, w_{2,i,j}, \cdots, w_{n,i,j})$$

其中：$w_{k,i}$ 和 $w_{k,i,j}$ 分别表示查询 q_i 和文档 d_i^j 在第 k 个词上的权重，则查询 q_i 和文档 d_i^j 的相关度 $R_{VSM}(d_i^j, q_i)$ 可以利用两个向量之间夹角 θ 的余弦值表示，即：

$$R_{VSM}(d_i^j, q_i) = \cos(\theta) = \frac{d_i^j \bullet q_i}{\left\| d_i^j \right\| \left\| q_i \right\|} = \frac{\sum_{k=1}^{n} w_{k,i,j} * w_{k,i}}{\sqrt{\sum_{k=1}^{n} w_{k,i,j}^2} * \sqrt{\sum_{k=1}^{n} w_{k,i}^2}}$$

权重有多种表示方法，这里介绍一种常用的 TF-IDF 模型，它是一种统计方法，用于评估相对于一个文档集，一个词在一个文档中的重要程度。词的重要性与它在文档中出现的频率成正比，与它在文档集中出现的频率成反比。如下表示：

$$w_{t,j}=tf_{t,d} \cdot \boldsymbol{idf}_t$$

其中：$tf_{t,d}$表示词 t 在文档 d 中出现的频率，即词频 TF（Term Frequency）。$\mathrm{idf_t}$ 表示词 t 在文档集合中出现的频率，即逆向文档频率 IDF（Inverse Document Frequency）。

$\boldsymbol{idf}_t$有多种计算方法，其中的一种为：

$$\boldsymbol{idf}_t = \log \frac{|\boldsymbol{D}|}{|\{d \in \boldsymbol{D} \mid t \in d\}|}$$

其中：$\boldsymbol{D}$ 表示整个文档集合，丨$\boldsymbol{D}$丨表示整个文档集合中的文档的个数，$|\{d \in \boldsymbol{D} | t \in d\}|$表示文档集 $\boldsymbol{D}$ 中含有词 t 的文档的个数。

Okapi BM25 模型

Robertson 等在 Okapi 系统中提出了概率模型计算公式 Okapi BM25[Robertson 1995]，它综合考虑了词频、文档长度等因素对查询与文档相关度的影响，公式如下：

$$\mathrm{BM25}(D,Q) = \sum_{t \in Q} w^{(1)} \frac{(k_1+1)tf\,(k_{\mathrm{a}}+1)qtf}{(k+tf)(k_{\mathrm{a}}+qtf)} + k_2 |Q| \frac{avdl - dl}{avdl + dl}$$

其中：查询 Q 的长度是丨Q丨。对于 Q 中的一个词 t，tf 和 qtf 分别是词 t 在当前文档和查询中出现的次数。dl 和 $avdl$ 分别是当前文档的长度和文档集中文档的平均长度。K 的计算公式如下：

$$K = k_1 \left((1-b) + b \frac{dl}{avdl} \right)$$

$W^{(1)}$是词 t 本身的权重，一般称作 Robertson/Sparck Jones(RSJ)权重因子，计算公式如下：

$$W^{(1)} = \log \frac{(r+0.5)/(R-r+0.5)}{(n-r+0.5)/(N-n-R+r+0.5)}$$

其中 N 是集合中的文档总数，n 是出现词 t 的文档数，R 是与该查询相关的文档数，r 是相关文档中含有该词 t 的文档数。通常，在第一次检索时，因为缺少相关性信息，R 和 r 的取值为 0，权重因子公式就简化为：

$$W^{(1)} = \log \frac{N - n + 0.5}{n + 0.5}$$

从直观上，出现在少数文档中的词的重要性要高于在大多数的文档中都出现的词，$W^{(1)}$考虑的正是这个因素对查询与文档相关度的影响。

在公式中，k_1，k_2，k_3 和 b 是可以调整的参数，通过调整他们的值，可以改变词频、文档长度等对相关度大小的影响。如 k_1 主要是控制词频 TF 的影响，k_1=0 时，相关度的计算过程不考虑词频的影响。随着 k_1 的增大，词频对相关度的影响也增大。Okapi 通过实验，发现 k_2=0 是一个相对比较有效的值。

7.2.2 查询无关的排序模型

查询无关的排序模型，是指依照文档的重要程度对文档进行排序，而与具体的查询无关，如 PageRank 模型、HITS 模型、TrustRank 等。下面简单介绍 PageRank 模型和 HITS 模型。

PageRank 模型

PageRank 模型是一个基于链接分析的查询无关的排序模型，以 Google 的创始人之一 Larry Page 的姓命名，2001 年 9 月被授予美国专利，并被各大商用搜索引擎使用。它的基本思想是如果网页 A 存在一个指向网页 B 的链接，则表明 A 的所有者认为 B 比较重要，从而把 A 的一部分重要性得分赋予 B。公式如下：

$$PR(d_u) = \sum_{d_v \in \boldsymbol{B}_u} \frac{PR(d_v)}{U(d_v)}$$

其中：$\boldsymbol{B}_u$ 是所有包含指向网页 d_u 的链接的网页的集合，U(d_v)是表示网页 d_v 中含有的链入其他网页的链接数。从公式可以看出，d_u 的 PageRank 值 $PR(d_u)$取决于所有链入 d_u 的网页的 PageRank 的值。

对上述 PageRank 公式一个更好的解决方法是加入一个平滑的参数，即一个随机漫游在网络中的用户，它不是一定要沿着网页的链接对网络进行浏览，它可能以概率α随机访问到一个网页，因此 PageRank 的公式如下：

$$PR(d_u) = \alpha \sum_{d_v \in \boldsymbol{B}_u} \frac{PR(d_v)}{U(d_v)} + \frac{1-\alpha}{N}$$

其中：N 为网络中的网页数。

HITS 模型

HITS（Hyperlink-Induced Topic Search）模型也是一种基于链接分析的查询无关的排序模型，由 Jon Kleinberg 设计，根据一个网页的入度（指向此网页的超链接）和出度（从此网页指向别的网页）来衡量网页的重要性，HITS 对一个网页有两个权重值：Authority 值和 Hub 值。Authority 值是网页内容的权重，Hub 值是网页链接的权重。伪代码如下：

```
G := 网页集合
for 每个网页 p in G do
     // p.auth 是网页 p 的 Authority 值
     p.auth = 1
     // p.hub 是网页 p 的 Hub 值
     p.hub = 1
end for

function HubsAndAuthorities(G)
// 算法运行 k 词循环
for step from 1 to k do
       // 首先更新所有的 Authority 值
       for 每个网页 p in G do
            // p.incomingNeighbors 是链入网页 p 的所有网页的集合
            for 每个网页 q in p.incomingNeighbors do
                 p.auth += q.hub
            end for
       end for
       // 然后更新所有的 Hub 值
       for each page p in G do
            // p.outgoingNeighbors 网页 p 链出的网页的集合
            for 每个网页 r in p.outgoingNeighbors do
                 p.hub += r.auth
            end for
       end for
end for
```

HITS 和 PageRank 一样，都是基于网页链接分析的查询无关的排序模型，但是他们存在着如下的不同：（1）HITS 是在查询阶段运行的，即需要在线上运行；而 PageRank 是在索引阶段运行的，即可以在线下运行。（2）HITS 为每个文档计算两个值：Authority 和 Hub 值，而 PageRank 为每个文档只计算一个值。（3）PageRank

广泛地用于商业搜索引擎，而 HITS 由于需要线上运行等原因，较少被商用搜索引擎采用。

7.3 排序学习简介以及研究现状

7.3.1 排序学习简介

排序学习是信息检索与机器学习的一个交叉学科，所以排序学习模型与传统的机器学习模型存在着密切的联系。但排序学习是应用于信息检索，因此排序学习模型与传统的机器学习模型又存在着本质的不同。要了解排序学习问题，就要清楚排序学习模型与传统的机器学习模型之间的关系。

排序学习模型与传统的机器学习模型存在着紧密的联系。排序学习模型是经过特定的假设，将排序问题转化为一个传统的机器学习问题，这种假设的方式有如下 3 种：

（1）假设文档之间都是相互独立的，如果预测查询与文档是否相关，即二值判断：相关或者不相关，那么排序问题可以转化为分类问题（Classification）；如果预测查询与文档的相关度数值，那么排序问题可以转化为回归问题（Regression）。

（2）假设文档对之间都是相互独立的，预测的是文档对中哪个文档更相关，进而给出文档对的偏序关系，那么排序问题可以转化为成对分类问题（Pairwise Classification）。

（3）如果将文档的排序序列作为样例，假设只有最优的排序序列才是正例，那么排序学习问题可以转化为结构化输出预测问题（Structured Output Prediction）。此内容在 7.7 节排序学习模型的简介中将具体介绍。

排序学习有着信息检索的应用背景，因此排序学习问题不能通过传统的机器学习完全地解释清楚，下面从 3 个角度分析排序学习问题与传统的机器学习问题的不同：

（1）从查询的角度，查询不仅仅是一个关键词的组合，文档的排序也需要考虑他们与查询的相关度，因此查询影响着排序问题的逻辑结构。例如同样的文档集合，对于两个不同的查询，应该给出两种不同的排序，将与相应查询相关的文档排在各自序列的前面。

（2）从文档排序序列的角度，文档排序时，相互之间的顺序更重要，而文档预测的分类，或者评分，只是起辅助排序的作用。如两个文档的评分分别为 1,2 或者 1,10，这两种评分对于这两个文档的排序没有任何的影响，也就是说文档评分之间相对的大小关系更重要，而不需要关心具体每个文档的分数是多少。

（3）从文档在排序中位置的角度，排序靠前的文档的相关度对排序效果的影响，与排序靠后的文档的相关度对排序效果的影响，这两者是不一样的。排序位置越靠前的文档，对排序效果的影响权重越大。因为当用户使用搜索引擎时，往往只是浏览排序位置比较靠前的几条结果。

7.3.2　排序学习问题的研究现状

排序学习作为信息检索与机器学习的交叉学科，越来越受到人们的重视，其表现有以下 4 个方面：

（1）从学术论文的角度，信息检索方面的顶级会议和顶级期刊中，排序学习相关的文章正在逐年增多。例如信息检索顶级会议 SIGIR（International ACM SIGIR Conference on Research and Development in Information Retrieval），从 2007 年开始，专门设立了关于排序学习的 Session，相关的文章占每年收录文章总数的 8%以上，并且从 2007 年开始设立排序学习的 Workshop，聚集学术界和企业界的研究者共同讨论排序学习最新的研究方向和问题。

（2）从排序模型的角度，现已提出多种模型，例如 Ranking SVM[Herbrich 2000][Joachims 2002]、SVM-MAP[Yue 2007]、ListNet[Cao 2007]、ListMLE[Xia 2008]、AdaRank[Xu 2007]、Frank[Tsai 2007]等，从理论、应用等各个角度对排序学习问题进行研究。

（3）从公开数据集的角度（参见 7.6 节），从 2007 年至今，陆续推出了 LETOR 数据集、Microsoft Learning to Rank 数据集、Yahoo Webscope 数据集等，方便研究者从事排序学习方面的研究。

（4）其他方面，学术界和企业界正以不同的方式推动着排序学习这一研究领域快速地成长，如 2010 年 3 月 1 日，Yahoo 公司组织了 Learning to Rank Challenge [CHALLENGE]，在排序学习领域，提供了一个公平的竞技平台，设立了两个任务，为每个任务提供相应的数据集，供全世界所有感兴趣的研究团体或者个人公平角逐，并许诺每个任务的前 4 名以丰厚的奖金。比赛共历时 3 个月，于同年 5 月 31 日结束。

获奖者在 2010 年 6 月 25 日召开的国际会议 ICML(International Conference on Machine Learning)，Learning to Rank Challenge Workshop 上，展示了他们的工作。

7.4 排序学习模型的应用实例

到目前为止，已经介绍了传统的排序模型，排序学习模型与传统机器学习模型的关系，以及排序学习的研究现状。排序学习模型学习排序函数的过程是什么？这一节我们以 Ranking SVM[Herbrich 2000][Joachims 2002]为例，向读者介绍一个排序学习模型应用到具体评测数据集上的例子，感兴趣的读者，可以根据本节给出的步骤，自己完成排序模型从训练、验证参数，到最后评测的过程。

Ranking SVM 的工具包采用[RankSVM]，评测数据集采用本章 7.6.1 中介绍的 LETOR4.0 中 MQ2007 数据集。MQ2007 数据集的查询集合采用的是 TREC 2007 年 Million Query track 的查询集合，文档集合采用.gov2 的文档集合。

MQ2007 数据集按照 5 份交叉验证的方式组织，从本章 7.6.1 中的表格 7-2 可以看出，有 5 个文件夹，在每个文件夹下进行一次训练、验证和测试的过程，因此需要进行 5 次。一次训练、验证和测试的过程为：在训练集合上，利用 Ranking SVM 模型训练排序模型；在验证集合上，对 Ranking SVM 模型训练得到的排序模型进行验证，从而调整参数 C，在验证集合上排序性能最优的排序模型对应的参数 C，作为 C 的最佳取值，并把它对应的排序模型应用于测试集合；在测试集合上，对排序模型的性能进行评测，对排序结果采用评测函数 P@N，MAP 和 NDCG@N 进行评测，关于评测函数，请见本章 7.8.1 节，这里采用 LETOR 提供的评测脚本 Eval-Score.pl 对结果进行评测。

5 次训练、验证和测试过程中，5 个测试集合上评测函数的平均值，作为 Ranking SVM 模型在 MQ2007 数据集上排序性能的评测。

在 MQ2007 数据集上的 5 个文件夹下运行 Ranking SVM，这里限定参数 C 的取值从如下值中选取：0.00001, 0.00002, 0.00005, 0.0001, 0.0002, 0.0005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10。伪代码如下：

```
// $set_base_path 是数据集所在的目录
define $set_base_path
//分别运行 5 个文件夹
for $fold in 1, 2, 3, 4, 5 do
      //$set 是每个文件夹所在的目录
```

```
        $set = $set_base_path/Fold.$fold
        //$C 是 Ranking SVM 中的参数 C，从如下值中选取一个最优的
        for $C in 0.00001, 0.00002, 0.00005, 0.0001, 0.0002, 0.0005, 0.001,
        0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10 do
            //利用训练集合 train.txt 训练模型，得到模型 model.$C
            svm_rank_learn -c $C -e 0.001 -l 1 $set/train.txt $set/
            model.$C
            //利用模型 model.$C 计算验证集合 vali.txt 中每个文档的得分，vali.sco.
            $C 为分数文件，每一行为具体的分数，与 vali.txt 中每一行的文档一一对应
            svm_rank_classify $set/vali.txt $set/model.$C $set/vali.sco.$C
            //利用 perl 脚本 Eval-Score.pl 对 vali.sco.$C 进行评测，得到三种
            评测函数 Precision@N，NDCG@N 和 MAP 的值，具体值见 vali.pef.$C。
            最后一个参数 0 表示不输出每个查询上的评测函数值，只输出平均值
            perl Eval-Score.pl $set/vali.txt $set/vali.sco.$C $set/vali.
            pef.$C 0
        end for
```

从所有的 vali.pef.$C 中选出一个使某一评测函数值最大的$C 的值，并把它赋予$C_max，这里我们选择使 MAP 值最大的$C 值

```
        choose among all $set/vali.pef.$C and define $C_max
        //利用模型 model.$C_max 对测试集合计算每个文档的评分，评分文件为 test.sco
        svm_rank_classify $set/test.txt $set/model.$C_max $set/test.sco
        //利用 perl 脚本 Eval-Score.pl 对 vali.sco.$C 进行评测。最后一个参数 1 表
        示输出每个查询上的评测函数值，以及所有查询上的平均值
        perl Eval-Score.pl $set/test.txt $set/vali.sco.$C $set/test.pef 1
    end for
```

5 个测试集合上各个评测函数的平均值请见本章 7.8.2 给出的结果。在第一个文件夹下，当 *C* 值为 0.5 时，文件 vali.pef.0.5 对应的 MAP 值最大，为 0.47，文件内容如下，分别给出了 P@1~16，MAP 和 NDCG@1~16 的值。

```
precision:  0.463126843657817   0.433628318584071
      0.430678466076696  0.421091445427729   0.412389380530973
      0.405113077679449  0.398230088495575   0.396386430678466
      0.394296951819076  0.387020648967551   0.377849289353714
      0.372664700098328  0.366008622645791   0.361567635903919
      0.356145526057031  0.353244837758112

MAP:0.469700842372357

NDCG:   0.400196656833825   0.392576204523107   0.405605336669687
   0.410427438025904    0.411375293362948   0.417696690878787
   0.423971519308384    0.432771592566972   0.441049700011071
```

```
0.446167064330112    0.448059559591954    0.452744488266579
0.456825696661949    0.462615948733157    0.467762223521132
0.474510652855145
```

此时排序函数 $h_{RankSVM}$ 的公式如下，这里对系数取小数点后两位有效数字：

$$h_{RankSVM}=-0.21F_1-0.07F_2+0.01F_3+\cdots+0.15F_{44}-0.29F_{45}-0.02F_{46}$$

F_i 为第 i 个排序特征，i 的取值为 i=1,2,3,…,45,46，一共 46 个排序特征。这个线性排序函数就可以作为最终的排序函数用于对文档排序。

7.5 排序学习方法的框架

本节首先向读者介绍本章中使用的各个符号的定义，然后介绍排序学习方法的框架，最后给出排序学习方法现在的研究现状。

7.5.1 参数设置

含有 N_Q 个查询的查询集合 $\boldsymbol{Q}$，其中第 i 个查询 q_i 对应的文档集合为 d_j，人工对查询 q_i 与 d_j 中第 j 个文档 d_i^j 的相关度进行标注的结果为 l_i^j，查询 q_i 与文档 d_i^j 对应的由各种特征组成的向量为 $\boldsymbol{X}_i^j=\left[X_{i,1}^j,X_{i,2}^j,\cdots,X_{i,m(d_i^j)}^j\right]$。一般来说，上下标注同时出现在一个变量上时，下标注标识查询的 *ID*，上标注标识文档的 *ID*。表 7-1 中给出了具体的定义和说明。

表 7-1 参数的定义与说明

定　义	说　明
$q_{i\in\boldsymbol{Q}}$	查询集合 $\boldsymbol{Q}$ 中的第 i 个查询，N_Q 为 $\boldsymbol{Q}$ 中查询的个数
$d_i=\left\{d_i^1,d_i^2,\cdots,d_i^{n(q_i)}\right\}$	第 i 个查询 q_i 对应的文档列表，文档的个数是 $n(q_i)$
$l_i=\left\{l_i^1,l_i^2,\cdots,l_i^{n(q_i)}\right\}$	l_i^j 为第 i 个查询 q_i 对应的第 j 个文档 d_i^j 人工标注的结果，即 l_i^j 标识出 q_i 与 d_i^j 的相关度
$\boldsymbol{X}_i^j=\left\{X_{i,1}^j,X_{i,2}^j,\cdots,X_{i,m(d_i^j)}^j\right\}$	查询 q_i 与其文档 d_i^j 对应的特征向量，$m(d_i^j)$ 为特征的数目

至此，我们得到用于排序学习算法的数据集：$Set=\{q_i,d_i,l_i,X_i\}_{i=1}^{N_Q}$。

7.5.2　排序学习方法的框架

排序学习方法的框架如图 7-3 所示，分为训练阶段和测试阶段。在训练阶段，训练集合和验证集合是由查询、文档以及它们之间的排序特征、标注结果构成的，

其中 $q_1,q_1,\cdots,q_T$ 为 T 个查询，$\begin{pmatrix} d_i^1 & \boldsymbol{X}_i^1 & \mathrm{l}_i^1 \\ d_i^2 & \boldsymbol{X}_i^2 & \mathrm{l}_i^2 \\ & \vdots & \\ d_i^{n(q_i)} & \boldsymbol{X}_i^{n(q_i)} & \mathrm{l}_i^{n(q_i)} \end{pmatrix}$ 为第 i 个查询 q_i 对应的文档、排序特征以及标注结果，$n(q_i)$为文档的个数，$\boldsymbol{X}_i^j$ 为 l_i^j 分别是第 j 个文档 d_i^j 对应的排序特征序列和标注结果。基于这一训练集合和验证集合，利用基于排序学习模型的学习系统，在最小化损失函数的条件下，学习得到排序模型 $f(q,d,\omega)$。在测试阶段，测试集中的一个查询 q 以及它对应的文档$(d^1,?),(d^2,?),\cdots,(d^n,?)$，排序系统利用排序函数 $f(q,d,\omega)$对文档进行排序，得到排序结果 $\begin{pmatrix} d^{i1} & f(q,d^{i1},\omega) \\ d^{i2} & f(q,d^{i2},\omega) \\ & \vdots \\ d^{in} & f(q,d^{in},\omega) \end{pmatrix}$。

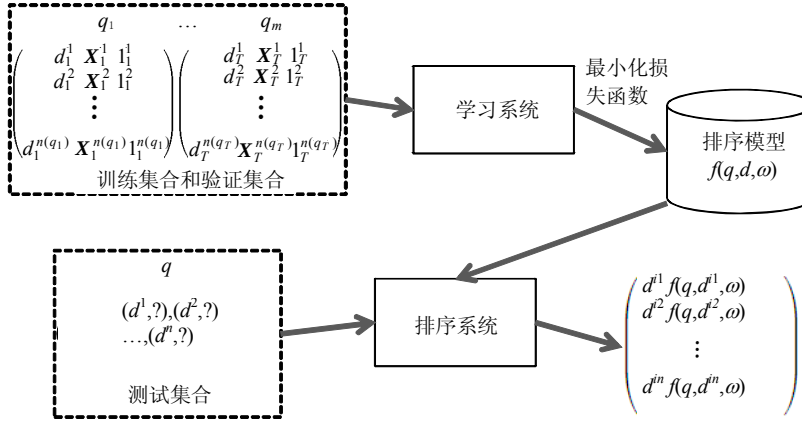


图 7-3　排序学习方法的框架

7.6 评测数据集

在各项研究中，当需要对比多种方法的优劣时，标准数据集（Benchmark Dataset）必不可少，它为各种方法的应用提供了一个共同的环境，信息检索的研究也不例外。有多种用于信息检索研究的数据集，如 SMART 数据集[SMART]，Virginia 数据集[VIRGINIA]，U. Tenn. Knoxville 数据集[LSI_CORPA]，TREC 数据集[TREC]，Glasgow 数据集[GLASGOW]等。为了对比各种排序学习模型的排序效果，也需要一个公用的数据集，本节向读者介绍在排序学习研究中经常使用的 3 种数据集：LETOR 数据集[Liu 2007][LETOR]，Microsoft Learning to Rank 数据集[MSLR]和 Yahoo Webscope 数据集[WEBSCOPE]。

7.6.1 LETOR 数据集

LETOR 是 **LE**arning **TO R**ank 的缩写，LETOR 数据集是由微软亚洲研究院提供的，专门用于排序学习的研究。截至目前，共有 4 个版本的数据集：LETOR1.0，LETOR2.0，LETOR3.0 和 LETOR4.0，数据集采用的格式和 SVMLight[RankSVM]一样。LETOR1.0 是 LETOR2.0 的 beta 版本，因此这里我们只介绍后 3 个版本。

LETOR2.0 是由信息检索中广泛使用的两个数据集构成的，它们是 OHSUMED 数据集和.gov 数据集。OHSUMED 数据集是医学数据集 MEDLINE 的子集，含有 348,566 篇文档，是从 1988—1991 年的 270 本医学杂志上获得的，共有 106 个查询，它曾用于 TREC 2000 年 Filtering Track。.gov 数据集含有 1,053,110 篇 HTML 文档，是在 2002 年 1 月份，从.gov 域下爬取的，它曾用于 TREC 2003 年和 TREC 2004 年 Web Track 下的主题提取任务（Topic Distillation Task），分别有 50 个和 75 个查询。

LETOR3.0 在 LETOR2.0 的基础上，在.gov 数据集上增加了 4 个数据集，曾分别用于 TREC 2003 年和 TREC2004 年的 Web Track 下的另外两个任务：Homepage ¯Finding Task 和 Named Page Finding Task。LETOR3.0 除包含 OHSUMED 数据集和.gov 数据集（包含 6 个数据集），还提供了更多的信息，用于构建更优的排序模型：为每个数据集抽取 meta 数据，它包含了查询的每个词在文档中的信息；为 OHSUMED 数据集提供文档相似度矩阵；为.gov 数据集提供了 Sitemap 和链接信息。

LETOR4.0 采用.gov2 的文档集合，约有 25M 篇文档，查询集合采用 TREC 2007

年和 TREC 2008 年 Million Query track 的查询集合，查询数量分别有 1692 个和 784 个。分 4 种形式的数据：Supervised ranking，Semi-supervised ranking，Rank aggregation 和 Listwise ranking。类似于 LETOR3.0，提供了 meta 数据，文档相似度矩阵，Sitemap 和链接信息。

表 7-2　5 份交叉验证

文 件 夹	训 练 集	验 证 集	测 试 集
Fold1	{*S*1,*S*2,*S*3}	*S*4	*S*5
Fold2	{*S*2,*S*3,*S*4}	*S*5	*S*1
Fold3	{*S*3,*S*4,*S*5}	*S*1	*S*2
Fold4	{*S*4,*S*5,*S*1}	*S*2	*S*3
Fold5	{*S*5,*S*1,*S*2}	*S*3	*S*4

应用数据集时一般是采用 5 份交叉验证的方式，如表 7-2 表示，将数据集随机平均分为 5 份：*S*1、*S*2、*S*3、*S*4 和 *S*5，3 份用于构建训练集，1 份用于构建验证集，1 份用于构建测试集。训练集是用来训练排序模型的；验证集用来调整某些排序学习模型的参数（如 Ranking SVM 模型中的参数 *C*），或者排序学习模型的循环次数（如 RankBoost 的循环次数 *T*）；测试集用来测试最终得到的排序函数的性能。按照表格 7-2 的方式，每一份数据集都有一次作为测试集，然后取排序模型在 5 个测试集上评测函数的平均值作为排序模型最终的排序效果。

7.6.2　Microsoft Learning to Rank 数据集

数据集中所有的查询、文档、标注结果都来自于商用搜索引擎 Microsoft Bing，共分为两个数据集：MSLR-WEB30k 和 MSLR-WEB10K。

MSLR-WEB30k 的查询数量超过 30,000 个，一个<查询，文档>对是由标注以及排序特征向量组成的。标注采用 5 级的方式，0 表示不相关，4 表示最相关，随着数字的增加，查询与文档的相关度增高。排序特征向量是由 136 个特征组成的，详情请见：[Feature List]。

MSLR-WEB10K 是从 MSLR-WEB30k 中随机采样 10,000 个查询得到的，相对于 MSLR-WEB30k 而言，只是数据规模上的差异。

数据依然以 5 份交叉验证的方式给出，用于排序模型的训练与评测。

7.6.3 Yahoo Webscope 数据集

Yahoo Webscope 数据集是在 Yahoo 举办的 Yahoo Learning to Rank Challenge 比赛上提供的数据集，比赛获奖者除了丰厚的奖金外，还被邀请参加第 27 届机器学习国际会议（ICML 2010）的专题讨论会（Workshop）。

比赛分为两个任务：一个是标准的排序学习任务（A standard learning to rank track），另一个是迁移学习任务（A transfer learning track），分别使用 Set1 数据集和 Set2 数据集。查询数、URL 数与特征数请见表 7-3。数据集采用的格式和 SVMLight [RankSVM]一样。

表 7-3　Yahoo Learning to Rank Challenge 数据集概况

	Set1			Set2		
	训练集	验证集	测试集	训练集	验证集	测试集
查询数	19,944	2,994	6,983	1,266	1,266	3,798
URL 数	473,134	71,083	165,660	34,815	34,881	103,174
特征数	519			596		

数据集采用 5 级标注：0,1,2,3,4。0 表示不相关，4 表示最相关，随着数字增大，相关度增加。其中只有当查询为导航类的需求，并且 URL 满足查询的导航类需求时，才被标注为 4。

7.7　排序学习模型简介

本节首先给出一个具体的实例，然后在此基础上，介绍现有排序学习方法的一种分类体系[Liu 2009]。

从机器学习的角度，模型有 4 个比较重要的概念：输入空间（Input Space）、输出空间（Output Space）、假设空间（Hypothesis Space）和损失函数（Loss Function）。输入空间是指模型所有可能的输入构成的集合，输出空间是指模型所有可能的输出构成的集合，假设空间是指模型生成的所有可能的函数构成的集合，损失函数是模型优化的目标，用来定义模型的输出与标准标注（Ground Truth Label）之间的差异。这里的标准标注，主要是用于训练时，作为训练样例的唯一标注，在这里把它归到

输入空间概念的范畴，在介绍输入空间时，将给出标准标注的定义。

根据这四个维度上定义的不同，现有的排序学习方法可以分为 3 类：Pointwise 方法，Pairwise 方法和 Listwise 方法。在介绍每种方法时，主要从这 4 个角度介绍每种方法，以及每种方法下具体的模型，而不涉及机器学习中的其他问题，如：以损失函数为优化目标，采用何种方式优化等。如果读者希望继续了解这方面的内容，请参考机器学习相关书籍和每种模型相关的参考文献。

7.7.1　实例

为了使读者在后文对每种方法进行介绍时，能对模型有更直观的理解，本节设计了一个实际的例子，这里我们利用查询集合 $\boldsymbol{Q}$ 中的一个查询 q_1，它对应 5 篇文档 $d_1=\left\{d_1^1,d_1^2,d_1^3,d_1^4,d_1^5\right\}$，即：$n(q_i)$=5，为每个<查询，文档>对抽取 3 个排序特征，q_1 与 d_1^j 对应的 3 个特征分别为：$x_{1,1}^j$，$x_{1,2}^j$，$x_{1,1}^j$。这个排序特征可以是 BM25，PageRank 等，并且是经过归一化的，即所有特征的取值范围都是[0,1]。标注采用的是 3 级标注的标准，对应的标注为 $l_1=\left\{d_1^1,d_1^2,d_1^3,d_1^4,d_1^5\right\}$，每个文档相应的排序特征的值以及标注如表 7-4 所示。

表 7-4　查询 q_1 对应的 5 个文档的排序特征以及标注的值

文　　档	特征 $X_{i,2}^j$	特征 $X_{i,2}^j$	特征 $X_{i,2}^j$	标注 l_i^j
d_1^1	0.489	0.013	0.024	0
d_1^2	0.772	0.247	0.007	1
d_1^3	0.005	0.767	0.011	0
d_1^4	0.696	0.871	0.008	2
d_1^5	0.694	0.398	0.003	1

7.7.2　Pointwise 方法

下面从输入空间、输出空间、假设空间、损失函数这四个方面介绍 Pointwise 方法。

Pointwise 方法的输入空间是由所有的单独文档对应的特征向量构成的集合，即：

$$Set_{Pointwise}^{InputSpace} = \left\{ \boldsymbol{X}_i^j \mid \forall q_i \in \boldsymbol{Q}, d_i^j \in d_i \right\}$$

在本节的例子中，特征向量是由 3 个排序特征构成的向量$[\boldsymbol{X}_{i,1}^j, \boldsymbol{X}_{i,2}^j, \boldsymbol{X}_{i,3}^j]$，因为特征值经过了归一化，即：$\boldsymbol{X}_{i,1}^j, \boldsymbol{X}_{i,2}^j, \boldsymbol{X}_{i,3}^j, \in [0,1]$，所以例子中的输入空间可以表示为：

$$Set_{Pointwise}^{InputSpace} = \left\{ [\boldsymbol{X}_{i,1}^j, \boldsymbol{X}_{i,2}^j, \boldsymbol{X}_{i,3}^j] \mid \forall q_i \in \boldsymbol{Q}, d_i^j \in d_i; X_{i,k}^j \in [0,1], k = 1,2,3 \right\}$$

从输入空间的定义可以看出，Pointwise 方法只考虑一个单独文档涉及的特征信息，不考虑文档与文档之间的关系。

Pointwise 方法的输出空间是由每个单独文档的所有可能的相关度构成的集合，例如相关度可以取实数集 $\boldsymbol{R}$，则输出空间为：

$$Set_{Pointwise}^{OutputSpace} = \boldsymbol{R}$$

本节例子中的输出空间也可以采用这样的定义方式。

3 种排序学习方法：Pointwise 方法，Pairwise 方法和 Listwise 方法，它们对应的标准标注，一般是借助查询与文档的相关度标识来定义的，而查询与文档的相关度标识有 3 种形式：（1）查询与单独文档的相关度，如本节的例子中，查询 q_1 与文档 d_1^1 的相关度标注 $I_1^1 = 0$；（2）查询对应的两个文档构成的文档相关度偏序对，如本节的例子中，查询 q_1 对应的两个文档 d_1^1 和 d_1^2，它们构成的偏序对，d_1^1, d_1^2 表示相对于查询 q_1，d_1^2 比 d_1^1 更相关，(d_1^2, d_1^1) 表示相对于查询 q_1，d_1^1 比 d_1^2 更相关；（3）查询对应的文档集合的相关度排序序列，如本节的例子中，对于查询 q_1，可能存在文档排序序列 $(d_1^1, d_1^2, d_1^3, d_1^4, d_1^5)$，表示相对于查询 q_1，从 d_1^1 到 d_1^5，相关度在减小。在第（2）和第（3）中提到的文档相关度偏序对和文档排序序列，有多种获得方式，如通过用户的点击日志[Joachims 2002]或者一个权威搜索引擎给出的文档排序结果等，不一定是通过本节例子中提到的相关度标注 l_i^j 获得的。

Pointwise 方法的标准标注，通过查询与文档的相关度标识的不同，定义分别如下：（1）如果给出文档 d_i^j 相关度的标注为 l_i^j，则 d_i^j 的标准标注 g_i^j 可以定义为：$g_i^j = \mathrm{l}_i^j$。在本节的例子中，对于查询 q_1，d_1^1 的标准标注为 $g_1^1 = \mathrm{l}_1^1 = 0$。（2）如果给出查询 q_i 下的文档相关度偏序对 (d_i^j, d_i^k)，一般没有一个直观的方法将这种偏序对的形式转化为 d_i^j 的标准标注。（3）如果给出查询 q_i 下的文档的排序序列 π_i，则文档

d_i^j 的标准标注可以通过一个关于π_i 的映射函数 $\Phi(\pi_i, d_i^j)$ 得到，这个映射函数有多种定义的方法，例如可以通过 d_i^j 在π_i 中的位置来定义 $\Phi(\pi_i, d_i^j)$ 的值，例如可以定义：$g_i^j = \Phi(\pi_i, d_i^j) = n(q_i) - \pi_i(d_i^j)$，其中 $\pi_i(d_i^j)$ 为 d_i^j 在排序序列π_i 中的位置。在本节的例子中，一种可能的排序序列为：$\pi_i = (d_1^1, d_1^2, d_1^3, d_1^4, d_1^5)$，则：

$$g_1^1 = \Phi(\pi_1, d_1^1) = n(q_1) - \pi_1(d_1^1) = 5 - 1 = 4$$

$$g_1^5 = \Phi(\pi_1, d_1^5) = n(q_1) - \pi_2(d_1^5) = 5 - 5 = 0$$

Pointwise 方法的假设空间是由以下所有的函数构成的集合：以一个文档对应的特征向量作为输入，输出为预测的这个文档的相关度，即：$Set_{Pointwise}^{HypothesisSpace} = F$。一般把完成这样功能的函数$f$，$f \in F$，称作评分函数，定义如下：

$$f : Set_{Pointwise}^{InputSpace} \to Set_{Pointwise}^{OutputSpace}, f \in Set_{Pointwise}^{HypothesisSpace}$$

根据评分函数，针对某一查询，可以对文档进行评分，并给出最终的排序结果。这个评分函数有多种定义的方法，如线性函数：

$$f(d_i^j) = \omega_0 + \sum_{k=1}^{m(d_i^j)} \omega_k X_{i,k}^j$$

在本节的例子中，线性函数的形式可以为：

$$f(d_i^j) = \omega_0 + \omega_1 X_{1,1}^j + \omega_2 X_{1,2}^j + \omega_3 X_{1,3}^j$$

Pointwise 方法的损失函数有多种定义方法，根据 Pointwise 方法建模时，利用回归（Regression），分类（Classification），或者顺序回归（Ordinal Regression），损失函数可以分别以回归损失（Regression Loss）、分类损失（Classification Loss）、顺序回归损失（Ordinal Regression Loss）的形式定义，分别在下面的具体排序学习模型中介绍。

Pointwise 方法的损失函数不考虑文档的内在依存关系，从而忽视了排序中文档的相对位置对损失函数值大小的影响。也就是说 Pointwise 方法没有考虑排序是相对于一个查询下的文档的排序，查询与文档、文档与文档是有依存关系的。因此从直观来讲，Pointwise 方法有其局限性。

根据 Pointwise 方法建模时使用的机器学习模型的不同，Pointwise 方法可以分为回归方法、分类方法、顺序回归方法，下面分别介绍这 3 种方法。

回归方法

排序问题，可以通过将查询与文档的相关度看成一个实数数值的方式，简化为一个回归问题，这类排序学习的方法叫做基于回归方法的排序学习方法，这类方法有[Cossock 2006][Fuhr 1989]等，下面简要介绍方法[Cossock 2006]。

对于查询 q_i 对应的文档集合 $d_i=\left\{d_i^1,d_i^2,\cdots,d_i^{n(qi)}\right\}$，文档集合中的每个文档对应的标准标注为 $g_i=\left\{g_i^1,g_i^2,\cdots,g_i^{n(qi)}\right\}$，标准标注可以直接通过查询与文档的人工标注 $\mathrm{l}_i=\left\{\mathrm{l}_i^1,\mathrm{l}_i^2,\cdots,\mathrm{l}_i^{n(qi)}\right\}$ 获得，定义为：$g_i^j=\mathrm{l}_i^j$。利用假设空间中的一个评分函数 f 对文档进行排序，评分函数 f 对文档 d_i^j 预测的评分为 $f(d_i^j)$。利用回归损失函数 $L(f;d_i^j,g_i^j)$ 度量评分 $f(d_i^j)$ 与文档对应的标准标注 g_i^j 之间的回归损失，定义如下：

$$L(f;d_i^j,d_i^j)=(g_i^j-f(d_i^j))^2$$

针对 7.7.1 节给出的例子，在 q_1 对应的 5 个文档上的损失为上述公式在每个文档上的值的加和，则在 q_1 对应的 5 个文档上的损失函数值为：

$$L(f;d_1,g_i)=\sum_{j=1}^{5}L(f;d_1^j,g_1^j)=\sum_{j=1}^{5}(g_1^j-f(d_1^5))^2$$

此时可以利用回归的方法以损失函数为优化的目标，学习评分函数。

方法[Cossock 2006][Fuhr 1989]存在同样的问题：标准标注 g_i^j 的数值大小是一个关于文档相关与否的判断，而不是数值上的判断，直接利用评分函数 f 来预测 g_i^j 的数值是有问题的。例如：$g_1^2=1$，则只有当预测评分 $f(d_1^2)$ 的值为 1 时，损失函数 $L(f;d_1^2,g_1^2)$ 的值才可能为 0，在其他情况下 $L(f;d_1^2,g_1^2)$ 均大于零，即存在损失。如：预测评分 $f(d_1^2)$ 的值为 2，即预测文档 d_1^2 为很相关的文档，此时损失函数的值为 $L(f;d_1^2,g_1^2)=1$；预测评分 $f(d_1^2)$ 的值为 0，即预测文档 d_1^2 为不相关的文档，此时损失函数的值为 $L(f;d_1^2,g_1^2)=1$。两种相反的情况却得到了相同大小的损失函数值。

分类方法

类似于将排序问题简化为一个回归问题，如果将查询与文档的相关度看成是对一个类别的标注，那么排序问题可以简化为一个分类问题。如果查询与文档的相关度采用 2 级标注标准，则排序问题可简化为一个二分类问题；如果相关度采用多级标注标准，则排序问题简化为一个多分类问题。因为基于分类方法的排序学习模型

不是将查询与文档的相关度看成一个实数数值，因此相对于基于回归方法的排序学习模型更为合理。这类方法有[Nallapati 2004][Li 2007]等，下面简要介绍方法[Li 2007]。

方法[Li 2007]采用多分类的方法解决排序问题，对于查询 q_i 对应的文档集合 $\boldsymbol{d_i}=\left\{d_i^1,d_i^2,\cdots,d_i^{n(qi)}\right\}$，文档集合中的每个文档对应的标准标注为 $\boldsymbol{g_i}=\left\{g_i^1,g_i^2,\cdots,g_i^{n(qi)}\right\}$，$g_i^j\in\{0,1,2,\cdots,K-1\}$，为 K 级标注，标准标注的获得方式与本节 Pointwise 方法中的回归方法相同。假设有一个多分类器，能够预测文档 d_i^j 的类别为 $\hat{g}_i^j$ 的概率：$P(\hat{g}_i^j=k)$，k=0, 1, 2,…, K，此时评分函数有多种定义方式，其中一种形式如下所示：

$$f(d_i^j)=\sum_{K=0}^{K-1}k\cdot P(\hat{g}_i^j=k)$$

损失函数 $L(\hat{g}_i^j,g_i^j)$ 用来描述文档 d_i^j 预测结果 $\hat{g}_i^j$ 与标准标注 g_i^j 的分类损失，可选的定义形式如下：

$$L(\hat{g}_i^j,g_i^j)=\sum_{K=0}^{K-1}\left(-\log\left(P(\hat{g}_i^j=k)\right)I(g_i^j\neq\hat{g}_i^j)\right)$$

其中：$I(.)$为指示函数（Indicator Function），定义为 $I(A)\begin{cases}1 & A\text{为真值}\\ 0 & A\text{为假值}\end{cases}$。

此时可以利用分类的方法以损失函数为优化的目标，学习评分函数。

顺序回归方法

顺序回归方法是在排序的过程中，考虑标准标注之间的关系，例如对于 K 级标准标注 $g_i^j\in\{0,1,2,\cdots,K-1\}$，顺序回归方法的目标是构造一个评分函数 f，使得 $f(d_i^j)$ 可以通过阈值 $b_0\leqslant b_1\leqslant\cdots\leqslant b_{K\text{-}1}=\infty$，被分到某一标准标注对应的类别中。这类方法有[Crammer 2002][Shashua 2002]，下面简要介绍[Crammer 2002]（简称 PRanking 模型）。

PRanking 的目标是找到最优的评分函数：

$$f(d_i^j) = \omega_0 + \sum_{k=1}^{m(d_i^j)\omega} {}_{\cdot k} X_{i,k}^j$$

即找到一个参数向量ω，当把文档d_i^j对应的特征向量X_i^j映射到ω上时，即可以利用阈值将文档分到某一个类别当中。通过迭代循环的学习过程，可以最终达到目标。对于本节的例子，由于ω_0对最终的排序没有影响，因此忽略ω_0，在第 t 次循环中，$\omega^T(\omega_1,\omega_2,\omega_3)$=(0,6,0,9,0,1)，阈值向量 b=(b_0,b_0,b_2)=(0,7,3,5,∞)，模型对文档d_1^4的预测值为：

$$f(d_1^2) = \sum_{k=1}^{3} \omega_k X_{1,k}^2 = 0.6*0.772 + 0.9*0.247 + 0.1*0.007 \approx 0.686$$

预测的标注采用 $\hat{g}_1^2 = \arg\min_k \left\{ f(d_1^2) - b_k < 0 \right\}$ 定义，则 $\hat{g}_1^2 = 0$，而文档d_1^2的标准标注为：$g_1^2 = 0$，正确的分类区间如图 7-4 所示，为：$b_0 < f(d_1^2) < b_2$，此时才有正确的预测类别 $\hat{g}_1^2 = 1$。如果减小 b_0 的值或者增大 $f(d_1^2)$ 的值，则可以得到正确的预测结果，模型通过更新向量ω^T和 b 完成这一工作。

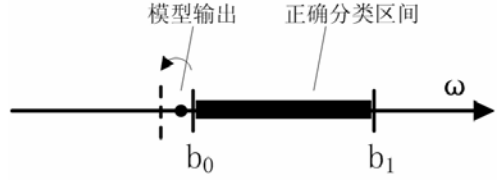


图 7-4　PRanking 例子

7.7.3　Pairwise 方法

下面从输入空间、输出空间、假设空间、损失函数四个方面介绍 Pairwise 方法。

Pairwise 方法的输入空间是由所有的文档对构成的集合，文档对中的每个文档是以排序特征序列的形式存在的，即：

$$Set_{Pairwise}^{InputSpace} = \left\{ (X_i^j, X_k^p) \mid \forall q_i \in Q, d_i^j \in d_i; \forall q_k \in Q, d_k^p \in d_k \right\}$$

在本节的例子中，每个特征值都是经过归一化处理的，因此输入空间为：

$$Set_{Pairwise}^{InputSpace} = \left\{([X_{i,1}^j, X_{i,2}^j, X_{i,3}^j], [X_{i,2}^p, X_{i,2}^j, X_{i,3}^j]) \mid \forall q_i \in Q, d_i^j, d_i^p \in d_i; X_{i,k}^j, X_{i,k}^p \in [0,1], k = 1,2,3\right\}$$

Pairwise 方法的输出空间是由文档对所有偏序的可能性构成的集合。一般来讲，输出空间为：$Set_{Pairwise}^{OutPutSpace} = \{+1, -1\}$。输出空间中的标准标注一般有以下 3 种定义的方法：

（1）如果给出的是相对于查询 q_i 的两个文档 d_i^j 和 d_k^p 对应的标注 l_i^j 和 l_i^p，则相对于查询 q_i，d_i^j 和 d_i^p 的标准标注 $g_1(d_i^j, d_i^p)$ 可以表示为：

$$g_i(d_i^j, d_i^p) = 2 \cdot I(I_i^j > I_i^p) - 1$$

其中 $I(.)$为指示函数（Indicator Function），定义为 $I(A)\begin{cases}1 & A\text{为真值} \\ 0 & A\text{为假值}\end{cases}$。在本节的例子中，对于查询 q_1 的两个文档 d_1^1 和 d_1^2，他们的标注为 $l_1^1 = 0$ 和 $l_1^2 = 1$，则相对于查询 q_1，d_1^1 和 d_1^2 的标准标注 $g_1(d_1^1, d_1^2)$ 为：

$$g_1(d_1^1, d_1^2) = 2 \cdot I(I_1^1 > I_1^2) - 1 = 2 \cdot I(0 > 1) - 1 = -1$$

表示相对于查询 q_1，文档 d_1^2 的相关度高于 d_1^1 。

（2）如果给出的是相对于查询 q_i 的两个文档 d_i^j 和 d_i^k 的偏序关系，则可以直接定义标准标注为 $g_i(d_i^j, d_i^k) = \begin{cases}+1 & \text{如偏序对为}(d_i^j, d_i^k) \\ -1 & \text{如偏序对为}(d_i^k, d_i^j)\end{cases}$。在本节的例子中，对于查询 q_1 的两个文档 d_1^1 和 d_1^2，如果给出相对于 q_1 的文档相关度偏序关系为 (d_1^1, d_1^2)，即 d_1^2 比 d_1^1 更相关，则 $g_1(d_1^1, d_1^2) = +1$。如果给出相对于 q_1 的文档相关度偏序关系为 (d_1^2, d_1^1)，即 d_1^1 比 d_1^2 更相关，则 $g_1(d_1^1, d_1^2) = -1$；

（3）如果给出查询 q_i 下的文档的排序序列 π_i，则可以定义文档 d_i^j 和 d_i^k 的标准标注为：

$$g_i(d_i^j, d_i^k) = 2 \cdot \mathrm{I}(\pi_i(d_i^j) > \pi_i(d_i^k)) - 1$$

其中 $\pi_i(d_i^j)$ 和 $\pi_i(d_i^k)$ 分别为 d_i^j 和 d_i^k 在排序 π_i 下的位置。在本节的例子中，对于查询 q_1，如果有排序序列 $\pi_1(d_1^1, d_1^2, d_1^3, d_1^4, d_1^5)$，从序列可以看出 d_1^1 比 d_1^5 更相关，d_1^1 比 d_1^5 的标准标注定义为：

$$g_1(d_1^1, d_1^5) = 2 \cdot \mathrm{I}(\pi_1(d_1^1) > \pi_1(d_1^5)) - 1 = 2 \cdot \mathrm{I}(1 > 5) - 1 = -1$$

Pairwise 方法的假设空间是由以下所有的函数构成的集合：以同一查询下的两个文档构成的特征向量对作为输入，输出是两个文档的文档相关度偏序关系，即：$Set_{Pairwise}^{HypothesisSpace} = H$。对于完成这样功能的函数 h，$h \in H$，定义为：

$$h : Set_{Pairwise}^{InputSpace} \rightarrow Set_{Pairwise}^{OutputSpace}, h \in Set_{Pairwise}^{HypothesisSpace}$$

一般情况下，可以简化为通过评分函数 f 来定义，例如对于 q_i 的两个文档 d_i^j 和 d_i^k，可以利用如下的形式来定义：

$$h(d_i^j, d_i^k) = 2 \cdot \mathrm{I}(f(d_i^j) > f(d_i^k)) - 1$$

Pairwise 方法的损失函数一般是基于文档逆序对数来定义的。文档逆序对是指函数 h 预测出的文档对的偏序关系，与这个文档对的标准标注不一致，如：对于查询 q_1 的两个文档 d_1^1 和 d_1^2，当 $h(d_1^1, d_1^2) * g(d_1^1, d_1^5) = -1$ 时，则相对于函数 h，文档对 (d_1^1, d_1^2) 是一个逆序对。

Pairwise 方法，一般情况下，是通过基于文档对的分类模型来训练排序模型，因此 Pairwise 方法的损失函数只是考虑文档对的偏序关系，而忽视了文档对中的两个文档在排序序列中的位置。如文档序列 $(d_1^1, d_1^2, d_1^3, d_1^4, d_1^5)$ 中，d_1^1, d_1^2 的偏序关系的重要程度，与 d_1^4, d_1^5 的偏序关系的重要程度是不一样的，因为搜索引擎用户往往只浏览排序靠前的结果，因此 d_1^1, d_1^2 的偏序关系的重要程度更高。

相对 Pointwise 方法而言，Pairwise 方法利用了文档对的相关度偏序关系，考虑了同一查询下文档与文档之间的关系，所以一般而言，Pairwise 方法的排序性能要优于 Pointwise 方法。

Pairwise 方法的工作有 Ranking SVM 模型[Herbrich 2000][Joachims 2002]，RankBoost 模型[Freund 2003]等，下面以 Ranking SVM 为例子，简要介绍 Pairwise 方法。

Ranking SVM 模型

Ranking SVM 模型[Herbrich 2000][Joachims 2002]是利用机器学习中的 SVM(Support Vector Machine) [Herbrich 2000]分类方法，面向文档对进行的分类，例

如对于查询 q_i 对应的文档对 (d_i^j, d_i^k)，对应的标准标注为 $g_i(d_i^j, d_i^k)=1$，这里采用的是线性的评分函数 $h(X_i^j)=\omega^T X_i^j$，ω为线性评分函数的系数向量，则 Ranking SVM 可以利用如下的数学公式表示：

$$\min \frac{1}{2}\|\omega\| + C\sum_{i=1}^{|Q|} \sum_{j,k:g(d_i^j,d_i^k)=1} \varepsilon_{i,j,\mathrm{k}}$$

约束条件为：$\omega^T(X_i^j - X_i^k) > 1 - \varepsilon_{i,j,k}$，其中 $\varepsilon_{i,j,k} \cdots 0, i = 1, \cdots, |Q|$

从公式可以看出，Ranking SVM 模型的优化目标和 SVM 模型是完全一样的，其中：$\frac{1}{2}\|\omega\|$ 用来控制模型的复杂程度，$\varepsilon_{i,j,k}$ 是松弛变量。Ranking SVM 模型采用的限制条件是文档对，这是和 SVM 模型不同的地方。这里采用文档对的 Hinge Loss 来定义损失函数。例如，对于查询 q_i 对应的文档对 (d_i^j, d_i^k)，如果相对于 q_i，文档 d_i^j 比文档 d_i^k 更相关，即标准标注为 $g(d_i^j, d_i^k)=1$，那么相对于边界（Margin）1，当 $\omega^T X_i^j$ 的值大于 $\omega^T X_i^k$ 时，则没有损失，否则损失为 $\varepsilon_{i,j,k}$。

7.7.4　Listwise 方法

下面从输入空间、输出空间、假设空间、损失函数四个方面介绍 Listwise 方法。

Listwise 方法的输入空间是由所有可能的文档集合（每一个文档集合对应着一个查询）构成的集合，其中的每个文档都是由特征向量来表示的。即：

$$Set_{Listwise}^{InputSpace} = \left\{ \left\{ X_i^1, X_i^2, \cdots, X_i^{n(q_1)} \right\} \mid \forall q_i \in Q \right\}$$

在本节的例子中，输入空间是：

$$Set_{Listwise}^{InputSpace} = \left\{ \left\{ [X_{i,1}^1, X_{i,2}^1, X_{i,3}^1], [X_{i,1}^2, X_{i,2}^2, X_{i,3}^2], \cdots, [X_{i,1}^{n(q_1)}, X_{i,2}^{n(q_1)}, X_{i,3}^{n(q_1)}] \right\} \mid \forall q_i \in Q \right\}$$

Listwise 方法的输出空间有两种形式：

（1）对于某些 Listwise 方法，输出空间可以是由文档集合对应的相关度集合构成的集合，即：

$$Set_{Listwise}^{OutputSpace} = \left\{ \left\{ r_i^1, r_i^2, \cdots, r_i^{n(q_1)} \right\} \mid \forall q_i \in Q, r_i^j \in R \right\}$$

其中：每一个相关度值 r_i^j，与文档 d_i^j 对应的特征向量 $[X_{i,1}^j, X_{i,2}^j, \cdots, X_{i,m(d_i^j)}^j]$ 一一对应。这种输出空间的标准标注，即 $\left\{X_i^1, X_i^2, \cdots, X_i^{n(q_i)}\right\}$ 的标准标注 $\left\{g_i^1, g_i^2, \cdots, g_i^{n(q_i)}\right\}$，它的获得方式，类似于 Pointwise 方法标准标注的获得方式，分别为 X_i^j 定义 g_i^j 即可。

（2）对于其他一些 Listwise 方法，输出空间是由所有文档的序列构成的集合，即：$Set_{Listwise}^{OutputSpace} = \left\{\pi_i^{OS} \mid \forall q_i \in Q\right\}$，此时如果给出的判断是文档序列$\pi_i$，则可以直接定义标准标注 g_i=π_i，否则需要利用等价置换的方法。

Listwise 方法的假设空间是由以下的所有函数构成的集合：以文档集合作为输入，输出是预测的文档集合中所有文档的相关度，或者文档集合中所有文档对应的排序序列，即假设空间为：$Set_{Listwise}^{HypothesisSpace} = \boldsymbol{H}_L$。一般情况下，对于函数 h，$h \in H_L$，可以利用评分函数 f 来实现 h，如对于输入 $\left\{X_i^1, X_i^2, \cdots, X_i^{n(q_i)}\right\}$，$h$ 的定义可以为：

$$h\left(\left\{X_i^1, X_i^2, \cdots, X_i^{n(q_i)}\right\}\right) = \text{sort}\left(\left\{f(\boldsymbol{X}_i^1), f(\boldsymbol{X}_i^2), \cdots, f(\boldsymbol{X}_i^{n(q_i)})\right\}\right)$$

其中：sort(B)是对集合 B 中的元素按大小进行排列，输出有序的序列。

Listwise 方法的损失函数与输出空间相对应，分为两种。当标准标注为 $g_i = \left\{g_i^1, g_i^2, \cdots, g_i^{n(q_i)}\right\}$，则损失函数可以定义为评测函数近似值或者上/下界。如果标准标注为 g_i=π_i，则损失函数用于度量输出 $h\left(\left\{\boldsymbol{X}_i^1, \boldsymbol{X}_i^2, \cdots, \boldsymbol{X}_i^{n(q_i)}\right\}\right)$ 与标准标注 g_i=π_i 的差别程度。

相比于 Pointwise 方法和 Pairwise 方法，Listwise 方法考虑了查询与文档、文档与文档之间的关系，Listwise 方法的损失函数可以很自然地考虑在同一查询下，文档在排序序列中的位置，因此从直观上，Listwise 方法更接近于排序任务的要求。

根据损失函数的不同，我们把 Listwise 方法分为：直接优化评测函数和最小化排序序列损失。

直接优化评测函数

一般是通过评测函数来区分排序模型的性能，从而以直接优化排序函数的方式来构造损失函数，是对排序学习最直接的优化方法，这类的排序学习模型有

SVM-MAP 模型，AdaRank 模型等，这里简要介绍 SVM-MAP 模型。

SVM-MAP 模型[Yue 2007]是一种利用结构化支持向量机[Tsochantaridis 2005]（Structural Support Vector Machine）对基于评测方法 AP（Average Precision）定义的可微上界进行优化的一种 Listwise 方法。

对于查询 q_i，文档集合 $\boldsymbol{d_i}=\left\{d_i^1,d_i^2,\cdots,d_i^{n(q_i)}\right\}$ 对应的标准标注为 $\boldsymbol{g_i}=\left\{g_i^1,g_i^2,\cdots,g_i^{n(q_i)}\right\}$，对于文档集合 d_i 任意错误的标注为 $\boldsymbol{g_{c(i)}}$，则 SVM-MAP 的优化目标可以利用如下公式表示：

$$\min\frac{1}{2}\|\omega\|+\frac{C}{|Q|}\sum_{i=1}^{|Q|}\varepsilon_i$$

约束条件为：$\forall g_{c(i)}\neq g_i,\omega^T\Psi(g_{c(i)},d_i)+1-\mathrm{AP}(g_{c(i)})-\varepsilon_i$

其中Ψ是映射函数，具体的定义这里就不给出了，感兴趣的读者可以参考文献[Yue 2007]。

最小化排序序列损失

最小化排序序列损失的模型有：ListNet 模型，ListMLE 模型等，这里简要介绍 ListNet 模型。

ListNet 模型[Cao 2007]是一个基于神经网络（Neural Network）模型的排序学习算法，它定义了一个基于置换概率的、面向排序序列的损失函数。最后得到的排序函数为线性模型，可以表示为ω，$f_\omega(d_i^j)$ 为算法对文档 d_i^j 的评分。为查询 q_i 对应的文档集合 $\left\{d_i^1,d_i^2,\cdots,d_i^{n(q_i)}\right\}$ 进行评分，得到评分向量：

$$f_i=\left[f_\omega(d_i^1),f_\omega(d_i^2),\cdots,f_\omega(d_i^{n(q_i)})\right]$$

利用[Plackett 1975]中提到的将置换（这里得到的评分向量可以看作是对原文档序列的一个置换）转换为概率的模型，即置换概率：

$$p_s(\pi)=\prod_{j=1}^{n(q_i)}\frac{\Phi(s_{\pi(j)})}{\sum_{k=j}^{n(q_i)}\Phi(s_{\pi(k)})}$$

其中π是作用于 $n(q_i)$个文档的置换，$\phi(.)$是一个递增、恒正的函数，S 为一个评分函数，即为文档计算得到它与查询相关度的评分，$S\pi(j)$表示在置换π中第 j 个位置的文档的评分。置换概率有一个性质，即如果 $f_i=\left[f_\omega(d_i^{j_1}),f_\omega(d_i^{j_2}),\cdots,f_\omega(d_i^{j_n(q_i)})\right]$满足

$$f_\omega(d_i^{j_1})>f_\omega(d_i^{j_2})>,\cdots,>f_\omega(d_i^{j_{n(qi)}})$$

则 $Ps(\langle 1,2,\cdots,n(q_i)\rangle)$的置换概率最大，$Ps\ (\langle n(q_i),n(q_i)-1,\cdots,2,1\rangle)$的置换概率最小。

一般来说，排序只是关心排在前 N 位的文档，例如一般人使用搜索引擎的时候，只是关心第一页的前 10 条结果（这里 N=10），所以可以定义置换概率的前 N 位的形式：

$$P_s(\pi)=\prod_{j=1}^{N}\frac{\phi(S_{\pi(j)})}{\sum_{k=j}^{n(q_i)}\phi(S_{\pi(k)})}$$

设 $L(I_i,f_i)$为损失函数，则利用交叉熵（Cross Entropy）来表示 $L(I_i,f_i)$，如下：

$$L(I_i,f_i)=-\sum_{\pi\in G_N}P_{\mathrm{l}}(\pi)\log(P_{f_{\mathrm{l}}}(\pi))$$

其中：G_N表示前 N 项文档置换的所有可能性。

7.7.5 3 种排序方法的对比

在 3 种方法的介绍中，读者可能注意到：3 种方法的假设空间中都提到了评分函数 f 的使用，但这并不意味着从本质上 3 种方法都可以归结为 Pointwise 方法。3 种方法的区分，是输入空间、输出空间、假设空间、损失函数 4 个维度上综合考虑的结果。

最后从模型排序效果与训练时间复杂度上对 3 种方法进行对比。

利用 3 种方法训练出的排序函数，一般来讲，Listwise 方法得到的排序函数的排序效果相对最优，Pairwise 方法次之。从模型训练时的时间复杂度上，当利用相同量级的数据集对模型进行训练时，一般情况下，Pointwise 方法时间复杂度最低，而 Listwise 方法的时间复杂度最高。

3 种排序学习方法的对比如表 7-5 所示。

表 7-5　3 种排序学习方法的对比

类　　别	Pointwise 方法	Pairwise 方法	Listwise 方法
输入空间	单独文档	文档对	文档集合
输出空间	单独文档的评分	文档对偏序关系	文档集合的评分或文档序列
假设空间	输入单独文档，输出单独文档的评分	输入文档对，输出文档对偏序关系	输入文档集合，输出文档集合的评分或者文档序列
损失函数	回归损失，或者分类损失，或者顺序回归损失	文档对的预测偏序关系与文档对标准标注的一致程度	评测函数的近似值或者上/下界，或者预测序列与标准标注序列的差别程度
训练得到的排序函数的性能	最差	居中	最优
训练时的时间复杂度	最低	居中	最高

7.8　排序学习模型性能比较

本节首先介绍搜索引擎中经常使用的对排序性能进行评价的评测函数，然后给出评测数据集上各种排序学习方法性能的比较。

7.8.1　评测方法

如何评测两个排序模型的优劣？一般的方法是：在一个公共的评测集上，每种排序学习模型利用评测集的训练集合和测试集合训练得到排序函数，然后排序函数对测试集合上的查询对应的文档进行排序，并利用评测函数对排序结果进行评测，最后取测试集合上所有查询的评测值的平均值，作为度量排序模型性能的指标。

现在提出了很多种用于搜索引擎排序性能评测的评测方法，这里只是介绍 4 种经常使用的评测函数：P@N，MAP，NDCG@N，ERR。

为了对比这 4 种评测函数，对于某一查询的 10 个文档，采用 5 级标注：0,1,2,3,4。0 表示不相关，4 表示最相关，随着数字的增加，查询与文档之间相关度增大。两个

排序函数分别对 10 个文档进行排序，排序结果如下：

$Ranking_1$=<3,4,2,0,1,3,0,1,0,1>
$Ranking_2$=<4,2,3,0,1,3,0,1,0,1>
$Ranking_3$=<4,2,3,3,1,0,0,1,0,1>

其中：第一个排序 $Ranking_1$ 对文档的排序结果为<3,4,2,0,1,3,0,1,0,1>，表示排在第一位的文档的标注为 3，排在第二位的文档的标注为 4，以此类推；第二个排序 $Ranking_2$ 相对于第一个排序，第一个文档和第二个文档交换了位置，然后第二个文档和第三个文档交换了位置；第三个排序 $Ranking_3$ 在第二个排序的基础上，交换第四个文档和第六个文档的位置。

下面分别介绍 4 种经常被使用的评测方法：P@N、MAP、NDCG@N、ERR，并分别给出它们对 $Ranking_1$、$Ranking_2$ 和 $Ranking_3$ 的评测结果。

7.8.1.1 P@N

p@N 是 Precision@N 的缩写，用来度量一个查询对应的文档排序中，前 N 个结果的准确率，当计算一个查询 q_i 对应的文档排序的 $P@N$ 的值时，它的定义如下：

$$P@N = \frac{1}{N}\sum_{j=1}^{N} \mathrm{rel}(d_i^j)$$

其中：$\mathrm{rel}(d_i^j) = \begin{cases} 1 & \text{如果} d_i^j \text{和查询是相关的} \\ 0 & \text{其他情况} \end{cases}$，一般情况下 N 的取值为 $N = 1, 2, \ldots, 10$。实际应用中，使用测试集合上所有查询对应的 P@N 的平均值作为排序性能的评价指标。

当标注结果为多级标注的时候，如本节中给出的例子：$Ranking_1$、$Ranking_2$ 和 $Ranking_3$ 采用的是 5 级标注的标准，这时根据具体的情况，有很多种定义的方法，例如这里可以定义：$\mathrm{rel}(d_i^j) = \begin{cases} 1 & d_i^j \text{的标注} \geqslant 2 \\ 0 & \text{其他情况} \end{cases}$。这时三个序列变为：

$Ranking_1$=<1,1,1,0,0,1,0,0,0,0>$_{2\text{级标注}}$
$Ranking_2$=<1,1,1,0,0,1,0,0,0,0>$_{2\text{级标注}}$
$Ranking_3$=<1,1,1,1,0,0,0,0,0,0>$_{2\text{级标注}}$

$Ranking_1$ 和 $Ranking_2$ 对应的 2 级标注的结果是一样的，那么 $Ranking_1$ 和 $Ranking_2$ 对应的 P@1~10 的值也一定是一样的，这里只给出 $Ranking_1$ 和 $Ranking_3$ 的 P@1~10 的结果，如表 7-6 所示。从结果中可以看出，排序 Ranking3 优于 Ranking1

和 Ranking_2。

表 7-6　排序 Ranking_1 和 Ranking_3 在 P@1~10 上的对比结果

排序名称	P@1	P@2	P@3	P@4	P@5	P@6	P@7	P@8	P@9	P@10
Ranking_1	1	1	1	0.75	0.6	0.67	0.57	0.5	0.44	0.4
Ranking_3	1	1	1	1	0.8	0.67	0.57	0.5	0.44	0.4

7.8.1.2　MAP

准确率和召回率是信息检索中两个重要的概念。准确率是用来衡量排序函数的查准率，召回率是用来衡量排序函数的查全率，评测方法 P@N 只考虑了准确率，并没有考虑一个查询中相关文档的数目对排序性能的影响，即没有考虑召回率对排序性能的影响。

评测方法 MAP（Mean Average Precision）是综合考虑准确率与召回率的一个评测函数，假设测试集合对应的查询集合为 $\boldsymbol{Q}$，利用 MAP 对排序函数在测试集合上的排序结果进行评测，它的定义如下：

$$\text{MAP}=\frac{1}{|\boldsymbol{Q}|}\sum_{i=1}^{|\boldsymbol{Q}|}\text{AP}_i=\frac{1}{|\boldsymbol{Q}|}\sum_{i=1}^{|\boldsymbol{Q}|}\frac{1}{R_i}\left(\sum_{j=1}^{n(q_i)}\text{rel}_i(d_i^j)\bullet(\text{P@j})_i\right)$$

其中：$|\boldsymbol{Q}|$是集合 $\boldsymbol{Q}$ 中查询的数目，R_i 是指第 i 个查询的相关文档的数目，$n(q_i)$ 是指查询 q_i 对应的文档总数，查询 q_i 对应的文档排序序列中，排在第 j 位的文档为 d_i^j，$\text{rel}_i(d_i^j)$ 和$(\text{P@j})_i$分别是相对于查询 q_i：$\text{rel}(d_i^j)$ 和 P@j 的值。

当 $\text{rel}_i(d_i^j)$ 和 P@j 的定义如 7.8.1.1 节所示时，Ranking_1 和 Ranking_2 在 2 级标注的情况下是一样的，Ranking_1 和 Ranking_2 对应的 MAP 的值也是一样的，因此只给出 Ranking_1 和 Ranking_3 的结果，如表 7-7 所示。从结果中可以看出，和 P@1~10 的结论一样，排序 Ranking_3 优于 Ranking_1 和 Ranking_2。MAP 和 P@1~10 都不能区分这两个排序函数的性能，它们都只考虑“相关”与“不相关”这样的两级标注，不考虑 3 级以上的多级标注对排序性能的影响。

表 7-7　排序 Ranking_1 和 Ranking_3 在 MAP 上的对比结果

排序名称	MAP
Ranking_1	0.92
Ranking_3	1

7.8.1.3 NDCG@N

对于查询对应的文档排序结果，直观而言，文档排得越靠前，被用户浏览的可能性就越大。在 3 级以上的标注标准下，2 级标注标准下的“相关”文档也可以被分为多个级别，因此在评测的过程中，相关度越高的文档，排得越靠前，评测函数应该给予更高的权重。

评测函数 NDCG（Normalized Discount Cumulative Gain）[Jarvelin 2000][Jarvelin 2002]综合考虑了多级标注中，文档的相关度以及文档的位置对排序性能的影响，它的定义如下：

$$\mathrm{NDCG@N}=\frac{\mathrm{DCG@N}}{Z_N},\ \mathrm{DCG@N}=\sum_{j=1}^{N}\begin{cases}2^{\mathrm{rel}(d_i^j)}-1 & j=1\\ \dfrac{2^{\mathrm{rel}(d_i^j)}-1}{\log(j)} & j>1\end{cases}$$

其中：Z_N 为结果序列在最优的情况下的 DCG@N 的值。$\mathrm{rel}(d_i^j)$ 的定义和评测函数 P@N、MAP 中的定义不同，它可以直接取标注的值，即：$\mathrm{rel}(d_i^j)=l_i^j$，在本节的例子中，$\mathrm{rel}(d_i^j)$ 可以取 0,1,2,3,4。一般 N 的取值为 N = 1, 2, …, 10。

$\mathrm{Ranking}_1$，$\mathrm{Ranking}_2$ 和 $\mathrm{Ranking}_3$ 对应的 NDCG@1~10 的值如表 7-8 所示。从结果中可以看出，对于排序序列 $\mathrm{Ranking}_1$ 和 $\mathrm{Ranking}_2$，之前提到的两种评测方法 P@1~10 和 MAP 不能区分这两个排序函数的性能，但是通过评测方法 NDCG 可知：在评测值 NDCG@1 上，$\mathrm{Ranking}_2$ 优于 $\mathrm{Ranking}_1$；在评测值 NDCG@2~10 上，$\mathrm{Ranking}_1$ 优于 $\mathrm{Ranking}_2$。有的时候，我们固定 N 的值，来比较排序函数性能的优劣，如当 N=5 时，可以看出排序性能从优到劣的排序是：$\mathrm{Ranking}_3$，$\mathrm{Ranking}_2$，$\mathrm{Ranking}_1$。

表 7-8　排序 $\mathrm{Ranking}_1$、$\mathrm{Ranking}_2$ 和 $\mathrm{Ranking}_3$ 在 NDCG@1～10 上的对比结果

排序名称	NDCG@1	NDCG@2	NDCG@3	NDCG@4	NDCG@5
$\mathrm{Ranking}_1$	0.47	1	0.90	0.86	0.86
$\mathrm{Ranking}_2$	1	0.82	0.85	0.80	0.81
$\mathrm{Ranking}_3$	1	0.82	0.85	0.93	0.93
排序名称	NDCG@6	NDCG@7	NDCG@8	NDCG@9	NDCG@10
$\mathrm{Ranking}_1$	0.94	0.93	0.94	0.94	0.95
$\mathrm{Ranking}_2$	0.89	0.88	0.89	0.89	0.90
$\mathrm{Ranking}_3$	0.92	0.91	0.92	0.92	0.93

7.8.1.4　ERR

评测方法 ERR（Expected Reciprocal Rank）是一个基于级联模型（Cascade Model）的评测方法，之前提到的三种评测函数，都认为同一查询下的文档之间是独立的，但是 ERR 考虑了排序靠前的文档的相关度对后续文档的影响。公式如下：

$$\mathrm{ERR}=\sum_{j=1}^{n(qi)}\frac{1}{j}\mathrm{R}(\mathrm{rel}(d_i^j))\prod_{k=1}^{j-1}(1-\mathrm{R}(\mathrm{rel}(d_i^k))),\mathrm{R}(\mathrm{rel}(d_i^j))=\frac{2^{\mathrm{rel}(d_i^j)}-1}{2^{\mathrm{rel}(d_i^j)\max}}$$

其中：$\mathrm{rel}(d_i^j)$ 表示第 j 篇文档与查询的相关度，和 7.8.1.3 节中的定义一样。$\mathrm{rel}(d_i^j)_{\max}$ 表示所有标注中相关度最高的标注，在本节的例子中，$\mathrm{rel}(d_i^j)_{\max}=4$。

$\mathrm{Ranking}_1$，$\mathrm{Ranking}_2$ 和 $\mathrm{Ranking}_3$ 对应的评测方法 ERR 的值如表 7-9 所示。从结果中可以看出排序性能从优到劣的排序是：$\mathrm{Ranking}_3$，$\mathrm{Ranking}_2$，$\mathrm{Ranking}_1$，并且 $\mathrm{Ranking}_3$ 和 $\mathrm{Ranking}_2$ 的排序性能差别很小。

表 7-9　排序 $\mathrm{Ranking}_1$、$\mathrm{Ranking}_2$ 和 $\mathrm{Ranking}_3$ 在 ERR 上的对比结果

排序名称	ERR
$\mathrm{Ranking}_1$	0.71
$\mathrm{Ranking}_2$	0.953
$\mathrm{Ranking}_3$	0.954

7.8.2　排序模型性能的比较

不同的排序模型，在不同的数据集上，利用不同的评测函数进行评测，它们的优劣关系可能发生变化。下面我们主要是针对 LETOR 上给出的排序模型，对排序模型在特定数据集，特定评测函数下做一个对比。

我们利用 LETOR4.0 数据集，对比各个方法的性能。这里给出的方法有：Pointwise 方法：BordaCount；Pairwise 方法：RankSVM-Struct，RankBoost；Listwise 方法：ListNet，AdaRank-NDCG，AdaRank-MAP。分别在 LETOR4.0 的 MQ2007 和 MQ2008 两个数据集上。

在 MQ2007 数据集上的实验结果如图 7-5，图 7-6，图 7-7 所示。从图中可以看出，3 种方法，在两个数据集上呈现相同的趋势：Pairwise 方法和 Listwise 方法好于

Pointwise 方法，而 Pairwise 方法和 Listwise 方法在数据集 MQ2007 上具有可比性；在 MQ2008 数据集上也呈现相同的趋势，如表 7-10 所示，这里只给出 P@1,3,5，MAP 和 NDCG@1,3,5 的对比结果。

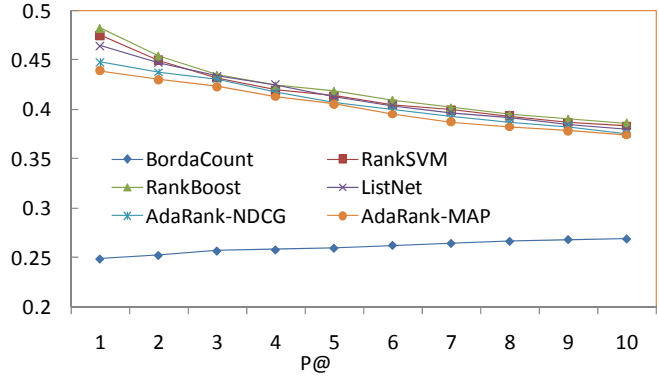


图 7-5　排序学习方法在 MQ2007 数据集上 P@1～10 的对比实验结果

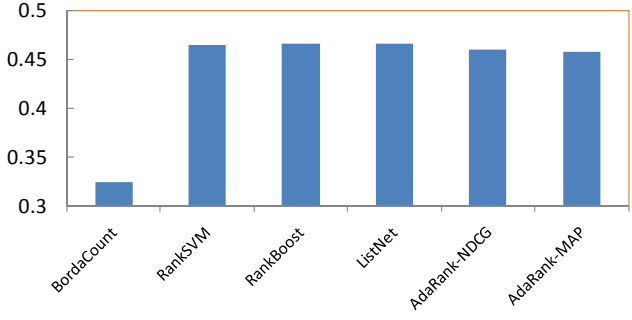


图 7-6　排序学习方法在 MQ2007 数据集上 MAP 的对比实验结果

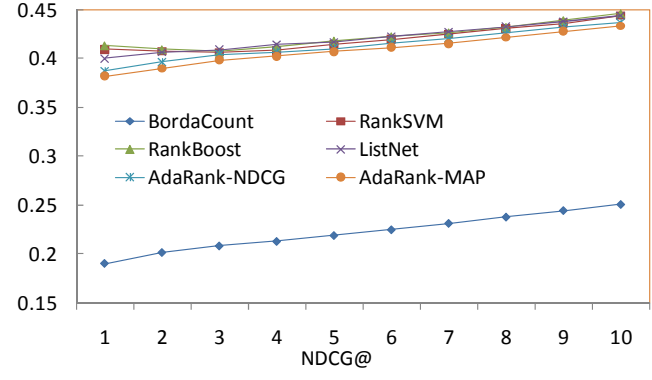


图 7-7　排序学习方法在 MQ2007 数据集上 NDCG@1~10 的对比实验结果

表 7-10　排序学习方法在 MQ2008 数据集上 P@1,3,5，MAP 和 NDCG@1,3,5 的对比实验结果

	P@1	P@5	P@10	MAP	NDCG@1	NDCG@5	NDCG@10
BordaCount	0.297	0.290	0.223	0.395	0.237	0.371	0.169
RankSVM	0.427	0.347	0.249	0.470	0.363	0.470	0.228
RankBoost	0.458	0.340	0.249	0.478	0.386	0.467	0.226
ListNet	0.445	0.343	0.248	0.478	0.375	0.475	0.23
AdaRank-NDCG	0.452	0.345	0.245	0.482	0.383	0.482	0.231
AdaRank	0.443	0.345	0.245	0.476	0.375	0.479	0.229

7.9　排序学习的研究方向

从本章 7.3.2 节介绍的排序学习的研究现状可以看出，排序学习问题越来越受到学术界和工业界的重视，在所有的研究当中，除了在本章第 7.7 节中介绍的对排序学习模型的研究外，还有很多其他的研究方向。本节介绍其中的部分工作，由于篇幅限制，不能详细介绍，感兴趣的读者可以通过相关的参考文献进一步了解。

7.9.1　标准标注的自动构建

本章 7.6 节中介绍的评测数据集，都包含查询与文档之间的相关度标注，有的是利用 2 级标注标准来标识相关度，如 LETOR2.0 中的.gov 数据集；有的是利用多级标注标准来标识相关度，如 Yahoo Webscope 数据集采用的是 5 级标注标准。这些标注的获得需要耗费大量的时间和人力，而查询与文档的相关度是用于构建排序学习中的标准标注，因此如何自动获得标准标注成为人们研究的一个热点。这类的工作有[Joachims 2002][Agichtein 2006]等。

7.9.2　排序特征

排序特征是构成排序函数的重要组成部分，如排序特征 BM25、PageRank 和 HITS 等，它们是设计者智慧与经验的结晶。可以利用计算机，自动地构建排序特征吗？有很多的研究者尝试解决这一问题，如[Almeida 2007][Fan 2004]等。

7.9.3 半监督学习/主动学习

考虑到标注数据的获得需要耗费大量的时间和人力，如何更有效地利用未经过标注的数据成为业界关心的问题，已经有很多工作利用半监督学习（Semi-supervised Learning）的方法使用未标注的数据提高排序模型的性能，如[Duh 2008]等。

7.9.4 查询相关的排序模型

查询有很多的分类体系，如查询可以根据查询意图分为导航类查询、信息类查询和事务类查询。不同类别的查询在很多排序特性上都存在着差异，如导航类查询对应的相关的网页数目很少，而信息类查询对应的相关网页数目较多，因此为所有的查询都采用同一个排序模型是不合适的。如何确定查询的分类体系，为每一个分类得到更有效的排序模型一直以来是业界关心的问题，这类的工作有[Zha 2010]等。

7.9.5 利用用户行为特征

用户行为的特征是指用户和搜索引擎交互时的各种动作，如表 7-11 所示，这些特征包含大量的信息，如用户对排序结果的满意程度，用户认为哪条结果满足自己的需求等，如何利用这些用户行为的特征，改进排序模型成为当今研究的一个热点，相关的工作有[Richardson 2006]等。

表 7-11 用户行为分类

名　称	详　情
查询	用户向搜索引擎提交的查询
点击结果	搜索引擎返回给用户的结果中，被用户点击的结果，以及这些结果在返回结果中排序的位置
停留时间	用户从开始提交查询，点击返回结果，一直到离开搜索引擎的时间间隔
用户与浏览器的交互行为	当用户使用搜索引擎时，点击了浏览器的打印，保存到收藏夹，以及鼠标在浏览器上的移动轨迹等
查询重构	用户在搜索引擎上寻找自己需要的内容时，多次重新构造查询的过程

7.10　总结

在本章中，我们主要介绍了排序学习的相关内容，并以文档检索应用为例子，首先介绍了传统排序模型。传统的排序模型按照排序模型与查询的关系，分为查询相关的排序模型和查询无关的排序模型。然后从与传统机器学习关系的角度，简要介绍排序学习，分析了排序学习的研究现状，并给出排序学习模型从训练、验证到测试的一个具体实例。最后从排序学习的框架、评测数据集、分类体系、性能和研究方向 4 个方面介绍了排序学习。其中，分类体系是从输入空间、输出空间、假设空间和损失函数 4 个维度，将排序学习方法分为 Pointwise 方法、Pairwise 方法和 Listwise 方法。Pointwise 方法是将基于一个文档的排序问题，简化为回归问题、分类问题或者顺序回归问题；Pairwise 方法是将基于文档对的排序问题，转化为基于文档对的分类问题；Listwise 方法通过构建基于评测函数的损失函数或者基于文档序列的损失函数，直接对文档集合的排序进行优化。

回首过去，在排序学习领域，聚集了国内外学术界、企业界诸多的研究者，发表了大量的研究成果，由于篇幅有限，只能简要介绍其中的部分研究成果，如读者对这一领域感兴趣，可以沿着本章给出的参考文献，继续学习。

参考文献

[Agichtein 2006] E. Agichtein et al. Learning user interaction models for predicting web search result preferences. In SIGIR 2006.

[Almeida 2007] H. Almeida et al. A combined component approach for finding collection-adapted ranking functions based on genetic programming. In SIGIR 2007.

[Cao 2006] Y. Cao et al. Adapting ranking SVM to document retrieval. SIGIR 2006.

[Cao 2007] Cao, Z., Qin, T., Liu, T., Tsai, M., and Li, H. 2007. Learning to rank: from pairwise approach to listwise approach. In ICML 2007.

[CHALLENGE] http://learningtorankchallenge.yahoo.com/

[Chirita 2005] P.A. Chirita et al. MailRank: using ranking for spam detection. In CIKM 2005.

[Collins 2002] Collins, M. Ranking algorithms for named-entity extraction: boosting and the voted perceptron. In ACL 2002.

[Cossock 2006] D. Cossock et al. Subset ranking using regression. COLT 2006.

[Crammer 2002] K. Crammer et al. Pranking with ranking. In NIPS 2002.

[Dave 2003] K. Dave et al. Mining the peanut gallery: opinion extraction and semantic classification of product reviews. In WWW 2003.

[Duh 2008] K. Duh et al. Learning to rank with partially-labeled data. SIGIR 2008.

[Fan 2004] W. Fan et al. A generic ranking function discovery framework by genetic programming for information retrieval. Information Processing and Management, 2004.

[Feature List] http://research.microsoft.com/en-us/projects/mslr/feature.aspx

[Freund 2003] Y. Freund. An efficient boosting algorithm for combining preferences. In J. Mach. Learn. Res. 2003.

[Fuhr 1989] N. Fuhr. Optimum polynomial retrieval functions based on the probability ranking principle. ACM Transactions on Information Systems, 7(3):183–204, 1989.

[GLASGOW] http://ir.dcs.gla.ac.uk/resources/test_collections/

[Gyongyi 2004] Z. Gyongyi et al. Combating web spam with TrustRank. In VLDB 2004.

[Harrington 2003] E. F. Harrington. Online Ranking/Collaborative filtering using the Perceptron Algorithm. In ICML 2003.

[Herbrich 2000] R. Herbrich et al. Large Margin Rank Boundaries for Ordinal Regression. In Advances in Large Margin Classifiers, Liu Press, 2000.

[Jarvelin 2000] K. Jarvelin et al. IR evaluation methods for retrieving highly

relevant documents. In SIGIR 2000.

[Jarvelin 2002] K. Jarvelin et al. Cumulated Gain-Based Evaluation of IR Techniques. ACM Transactions on Information Systems, 2002.

[Joachims 2002] T. Joachims. Optimizing search engines using clickthrough data. In KDD 2002.

[LETOR] http://research.microsoft.com/en-us/um/beijing/projects/letor/

[Li 2007] P. Li et al. McRank: Learning to rank using multiple classification and gradient boosting. In NIPS 2007.

[Liu 2007] T.-Y. Liu, J. Xu, T. Qin, W. Xiong, and H. Li. Letor: Benchmark dataset for research on learning to rank for information retrieval. In SIGIR 2007.

[Liu 2009] T.-Y. Liu. Learning to Rank for Information Retrieval. In Foundation and Trends on Information Retrieval, Now Publishers, 2009.

[LSI_CORPA] http://www.cs.utk.edu/~lsi/corpa.html

[MSLR] http://research.microsoft.com/en-us/projects/mslr/

[Nallapati 2004] R. Nallapati. Discriminative models for information retrieval. In SIGIR 2004

[Pang 2005] B. Pang et al. Seeing Stars: Exploiting Class Relationships for Sentiment Categorization with Respect to Rating Scales. In ACL 2005.

[Plackett 1975] R. L. Plackett. The analysis of permutations. In Applied Statistics 1975.

[Qin 2010] T. Qin et al. LETOR: A Benchmark Collection for Research on Learning to Rank for Information Retrieval, Information Retrieval Journal, 2010.

[RankSVM] http://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html

[Richardson 2006] M. Richardson et al. Beyond PageRank: Machine Learning for Static Ranking. In WWW 2006.

[Robertson 1995] S. E. Robertson et al. Okapi at TREC-3. In TREC 1995.

[Salton 1975] G. Salton et al. A Vector Space Model for Automatic Indexing. Communications of the ACM, vol. 18, nr. 11, pages 613–620, 1975.

[SMART] ftp://ftp.cs.cornell.edu/pub/smart/

[Shashua 2002] A. Shashua et al. Ranking with large margin principles: Two approaches. In NIPS 2002.

[TREC] http://trec.nist.gov/data.html

[Tsai 2007] M. Tsai et al. FRank: a ranking method with fidelity loss. In SIGIR 2007.

[Tsochantaridis 2005] I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun, Large Margin Methods for Structured and Interdependent Output Variables. In JMLR 2005.

[VIRGINIA] http://fox.cs.vt.edu/VAD1/

[WEBSCOPE] http://webscope.sandbox.yahoo.com/

[Xia 2008] Xia, F., Liu, T., Wang, J., Zhang, W., and Li, H. 2008. Listwise approach to learning to rank: theory and algorithm. In ICML 2008.

[Xu 2005] J. Xu et al. Ranking definitions with supervised learning methods. In WWW 2005.

[Xu 2007] J. Xu et al. AdaRank: a boosting algorithm for information retrieval. In SIGIR 2007.

[Yue 2007] Y. Yue et al.A support vector method for optimizing average precision. In SIGIR 2007.

[Zha 2010] H. Zha et al. Ranking Specialization for Web Search: A Divide and Conquer Approach Using Topical RankSVM. In WWW 2010.

第 8 章　搜索引擎的性能调优

8.1 系统调优概述

搜索引擎在为用户提供搜索服务时，往往能够在 50ms 以内的时间返回海量数据中和查询最相关的若干结果，同时需要控制成本，因此“快”和“省”就成为一对矛盾。如何在有限的资源获得最大的效能成为性能调优的主要动力，优化的极致追求就是让每次搜索的成本尽可能低，同时让用户的体验尽可能好。

性能调优似乎并没有一个学术上的统一提法，Intel 提出解决性能缺陷的 3 个层次[intel optimization level]：系统级、应用级和微架构级，如表 8-1 所示：

表 8-1 Intel 的性能优化层次

次 序	优化级别	目 标	主要关注问题	获 益
1	高： 系统级	通过改善应用和整个软硬件的相互关系提升性能	网络 磁盘 内在访问	高
2	中： 应用级	通过改善应用本身的算法	进程/线程同步 堆竞争 线程的实现 API 的选择，库函数选择 数据结构	中
3	低： 微架构级	通过在特定的微架构级优化来改善应用性能	微架构调优 数据和操作的局限性（缓存的效率） 数据对齐 向量化/SSE	低

分类可能是站在厂商立场，因此涉及 CPU 的优化单独出来，并按照优化效果进行划分。笔者就个人经验，在本章中划分为硬件和操作系统层、业务算法层两个层次，如表 8-2 所示：

表 8-2 本书关于优化讨论的层次

次 序	优化层次	目 标	主要关注问题
1	硬件和操作系统层	通过熟悉程序运行的硬件和操作系统环境，进行特定环境下的开发和运营	CPU 缓存 内存 磁盘 网络

续表

次　序	优化层次	目　标	主要关注问题
2	业务算法层	通过具体业务，例如搜索引擎的具体特性，设定特定的优化方法	具体业务相关问题

系统层主要解决的是对机器硬件和操作系统的调优，调优的对象是运行的程序，要求程序可以最小的消耗，且在最短的时间执行完，因此，深入理解其运行的操作系统平台和硬件结构尤为关键。本书在系统层方面，主要介绍瓶颈识别和 CPU、芯片缓存、内存、磁盘及网络方面的调优，并举例说明。

业务层主要解决的是面向业务、面向具体问题的一些具体方法，最终优化的结果将在这一层得到实际的体现。例如对硬盘读取速度进行了优化，但若不结合业务就没有任何价值。业务层相关内容在此前章节均有提及，但不是本节的重点，本节仅从硬件和操作系统层面进行优化的探讨。

由于大部分搜索引擎都运行于 Linux 操作系统，因此本书不涉及其他操作系统。如果没有特别说明，均表示作用于 Linux 操作系统。

8.2　瓶颈识别

调优的前提是发现问题，对短板进行调优，因此问题发现的能力是调优的基本能力。常用的命令如表 8-3 所示：

表 8-3　常用的机器运行状况查询工具

命　令	主要功能
top	显示机器目前运行的进程，可以对任意字段进行排序（PID、age、time 等），便于查找资源消耗大户。每 5 秒自动刷新
vmstat	显示关于进程、内存、对换区、IO、CPU 等信息，可以支持定制频率的采样，便于研究动态数据
ps	显示用户查询进程相关的更详细的信息，例如线程 ID、优先级、RSS（Resident set size，进程工作集大小，不含对换区）
free	显示内存使用状况
iostate	显示 CPU 和磁盘子系统的统计信息，每秒 IO 请求数，每秒读取写的块数等。常用 iostate –x 查询更丰富的统计信息，例如每秒合并的读写请求数等
sar	用于周期性收集汇报系统包括 CPU 使用，内存使用等全面信息

续表

命　令	主要功能
pmap	显示进程使用内存的情况，从而判断内存是否是程序运行的主要瓶颈
netstat	显示几乎所有关于网络的情况
tcpdump	是一个 Linux 下的抓包工具，用于分析协议
strace	显示某个进程运行期系统调用的状况，还可以通过 strace -c <command>来分析命令运行过程中系统调用的时间开销
time	显示进程运行的真实时间，用户态时间和内核态时间等系统资源消耗情况

每个命令侧重点不同，使用方法复杂，因此详细了解每个命令的使用需要在工作中多积累，多查帮助，除此之外还可以使用一些常用的工具例如 vtune、vagrind 等。

不同程序优化的方向往往有很大不同，例如爬虫耗费的是网络 IO，分词主要耗费的是 CPU，索引耗费的是磁盘 IO，查询主要耗费的是内存，Page Rank 的计算主要耗费的是 CPU 等。一般大体分为计算密集型、IO 密集型和混合型 3 种，下一节将逐步展开这些细节，从大体上了解优化的一些具体方法。

8.3　涉及 CPU 的优化方法

对于计算密集型的程序，通常是对机器独占的，或者 CPU 独占的，此外，当前的服务器通常是 SMP 架构，即多核 CPU 共享内存和总线。一般来说有以下一些基本方法。

- 将其他程序关闭或者放在计算密集型程序运行期以外的时间运行，确保程序对机器的独占。
- 对于实时性要求不强的程序，通过 renice 降低其优先级，从而避免计算密集型进程对机器的过度使用，例如日志分析程序。renice 10 2010，将进程号 2010 的进程调整到优先级 10 上。
- 使用 taskset 命令将进程绑定到某个或某几个 CPU 核上（仅在多核 CPU 上有效），从而避免进程上下文的切换（TLB 等）。
- 将外部中断绑定到某个或某几个 CPU 核上，echo 03 > /proc/irq/19/

smp_affinity，该命令将中断 19 绑定在 3 号 CPU 核上。

- 编写流水线优化的代码，通过乱序和减少分支的方法提高流水线的通畅性。
- SMP 结构下，每核的计算空间要具有独立性，计算之间通过 lock-free 的 queue、ring buffer 等数据结构交互数据。
- 避免上下文频繁切换。
- 涉及芯片缓存的 False sharing 问题。

下面各小节将对以上方法依次展开。在介绍 CPU 的调优前先简单介绍 SMP（Symmetrical Multi-Processing），之后的很多优化问题都与该结构有关。SMP 简单地说就是多个 CPU 核，共享一个内存和总线，L1 cache 也叫芯片缓存，一般是私有的，即每个 CPU 核带一个，L2 cache 可能是私有的也可能是部分共享的，这种结构的好处是计算资源可以很容易地添加和裁撤，但遗憾的是在获得高性能的计算能力以后，内存全局上却仅有一个，锁的情况不可避免，数据同步的复杂性大大提高等各种各样的问题随之而来。

8.3.1　上下文切换问题（context switching）

进程是系统资源的最小管理单位，在多核环境下，进程运行的若干信息被存储在 CPU 的寄存器和相应的芯片缓存内，这些被存储在 CPU 寄存器内的信息被称作是进程运行的上下文。在某个进程发生切换时，其上下文被保存在内核模式的栈中（kernel mode stack），下一个运行的进程的上下文回填到寄存器中，然后被执行。上下文切换的状况可以用 vmstat 命令，system 的 cs 一列上查看。上下文切换的主要代价主要有 3 点：TLB flush，cache pollution 和 pipeline pollution。

TLB（Translation Lookaside Buffer）flush 是最主要的开销，TLB 是 CPU 的一小块 cache，用于建立虚地址到实地址的映射，提高计算的速度，在发生上下文切换时，被切换的进程的 TLB 被置换出芯片缓存，切换的进程的 TLB 被换入。

cache pollution 是其次的开销，进程运行时会将运行数据往较高的内存层次移动（L1 cache L2 cache 等），当发生中断时，中断处理程序会在其中充斥很多数据，在被切换的程序换回时，这些高层次内存中的数据已经面目全非，因此还得再次从低层次的内存中（主存）往较高层次的内存移动，一个典型的特征是在中断较多的情

况下，进程的 cache miss 的比率大大增加。

pipeline pollution 指的是 CPU 流水线被破坏的情况，和 cache pollution 类似。

8.3.2 中断和轮询

中断处理是一种低延时，但高成本的操作。中断一般由 IO 设备产生，例如网卡、键盘、鼠标等。由于某些设备（例如键盘）要求快速响应，因此中断处理程序会立即将 CPU 上正在运行的程序换下，立即执行，对于系统的实时性是有价值的，但是过多的中断会导致上下文切换，上下文切换会导致 cache pollution、TLB flush 和 pipeline pollution，这样就大大降低系统性能。

Linux 操作系统包含硬中断和软中断，硬中断一般来源于硬件设备，软终端一般来源于操作系统和进程自定义协议。中断可以在/proc/interrupts 内查看。在一个多核程序中，惯用法是将某种特定的频发中断绑定到特定 CPU 核上，例如，可以将网卡中断绑定到特定 CPU 核上，以将上下文切换的损害做到影响范围的可控，用 ifconfig eth1 来查看网卡中断号，假定为 169，则通过使用用命令 echo 02 > /proc/irq/169/smp_affinity，来将 169 号硬中断绑定到编号为 2 的 CPU 核上。有研究表明[Intel 2008]，在特定状况下，提升可达 24%。

轮询具有低成本，高延时（high latency）的特点。轮询是指操作系统定时查询设备的状态，当有数据 IO 时，进入收发处理程序。相当于系统对设备有计划地进行控制，这样 cache、TLB 和 pipeline 的代价就可以避免。对于轮询模式，最好的情况下等价于中断请求的效果，最坏的情况下为一个轮询的周期。如图 8-1 所示，4 条细线表示轮询的时间点，第一条粗线表示第一次 IO 数据产生，第二条粗线表示在第二次轮询后，该数据得到处理，可见这种情况是最坏的情况，下面依次是最好情况和平均情况，不再赘述。

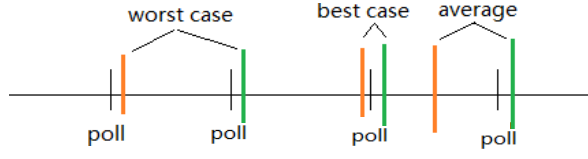


图 8-1 三轮询的延时不确定性

举个具体的例子，在大规模数据通信的过程中，网卡作为双向通信的接口，每有数据收发都会产生一次硬件中断，从而打断 CPU 正在执行的进程，发生一次上下

文切换，如果通信产生的包数量较大，且很频繁时，这种上下文切换的开销累积将很严重，将拖累整个系统性能，因此网卡设备一般都支持了 NAPI 的方式，对于通信过程中的第一个包，NAPI 会采用传统的方法发起一次中断请求，此后收发过程进入轮询模式（polling mode），只要在 DMA 的 ring buffer 中存在待收发的包时，收发过程不再产生新的中断，在处理最后一个包以后，DMA 的 ring buffer 被释放，重新进入中断模式，这样就有效地避免了上下文切换频繁带来的代价。

8.3.3　CPU 的 Affinity 问题

CPU 的 Affinity（亲和性）问题指的是在多核环境下，为了提高效能，进程或者线程的优先在某个特定处理器调度的策略。这样的好处在于处理器往往还在 cache 中保存着若干该进程的数据信息，一旦调入后，该处理器可以在 cache miss 很少的情况下继续执行，倘若调度到其他处理器，则需要重新载入大量数据到 cache 中，从而降低效能。

通常，操作系统会决定进程对 CPU 亲和性的程度，发挥 CPU 的最大利用率，同时兼顾进程的执行效率，但本质上 affinity 只是解决了 cache 的问题，并没有解决负载均衡的问题，在特定情况下，进程执行的工作集较大，且设置 affinity 后证明有效的情况下，可以将该进程绑定在某个或某组 CPU 上，从而改善性能，查询一个 CPU 的 Affinity 的方式：taskset -p [pid]。将一个进程绑定到一组 CPU 的方式：taskset -c [0,1..*N*] [pid]，代码可用 pthread_setaffinity_np 和 pthread_getaffinity_np 等函数。

举个例子，通常我们将 mysql 或者 oracle 这类工作集较大的程序做 CPU affinity 的操作。

8.3.4　流水线问题

精简指令计算机（RISC）处理器的设计目标就是平均每个时钟周期执行一条指令，虽然 RISC 是精简的，但执行一条指令还需要多个步骤（需要多个时钟周期），怎么可能做到每周期执行一条指令呢？考察这样一个简单的指令 mov([ebx],eax)需要经过的步骤如下：

（1）取指令的操作码。

（2）更新 EIP 寄存器，将其值改为紧随操作码之后的字节的地址。例如指令流

中 mov 的下一个指令是 jnz，则 EIP 的值指向 jnz 这个指令的地址。

（3）对操作码进行解码，得到指定的指令（mov 必须翻译成机器可以执行的指令）。

（4）从原寄存器中取值（从 ebx 寄存器中取值）。

（5）将值存储到目标寄存器中（写入 eax 寄存器中）。

按照上面提到的这个指令执行的过程，假定设计这样一个 6 级流水，定义为：

（1）取操作码

（2）解码操作码（并预取操作数）

（3）计算有效地址

（4）获取地址值

（5）计算

（6）存入结果

来看这样一个时钟周期和指令执行的过程，假定都可以并行执行（后面我们会讨论流水线停滞），如图 8-2 所示。

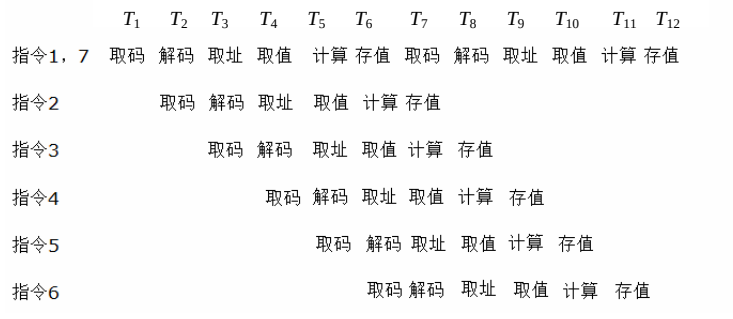


图 8-2　6 级流水线执行的动态情况

理想的情况下，在 *T*1 到 *T*6 这 6 个时间段中，流水线被装满，前 6 个指令依次装入流水线。从生产结果的情况看，从 *T*6 开始流水线做完了指令 1，*T*7 做完了指令 2……*T*11 做完了指令 6，*T*12 时刻完成了指令 7，产生了轮回。这样就呈现出（从头 *T*6 开始）每周期执行一条指令的态势，这一切都是并发指令带来的结果。

但不难理解在执行某个指令时，必须要能够正确地猜测出下一个指令的位置，才有可能进行正确的预取，一次错误的猜测会导致整个流水线毁掉，重新初始化。另外，还要注意流水线的阶段数并不是我这里举的简单的 6 段，不同硬件划分不同，流水越深预取失败的代价就越大。

通常情况下，流水线停滞主要由两方面原因导致，cache 不命中和数据依赖，具体解释如下：

（1）cache 不命中，一方面因为芯片 cache 的容量有限，另一方面也由程序局部性不强导致，这里不进行展开。一般向流水线中插入暂停周期来进行等待。

（2）数据依赖导致的停滞，通常由编译器通过乱序（Out of Order）来解决。将无数据依赖的指令提前补充到流水线中，相当于提前计算，有数据依赖的指令推迟执行。

对于开发工程师而言，主要考虑的是流水线重置问题，这主要由分支指令引起。当出现分支指令时，一旦指令预取发生错误，整个流水线需要重置，影响极大。去除分支指令有很多种方法，这种去除分支指令的代码也叫做 branch-free code。以下通过举例来给出一个直观的印象，实际工作中可以逐渐积累。

减少分支的一些惯用法：

（1）分支结果数组化。

```
for(;;)  {
     if(mode == 0)  sum += 12;
     else if(mode == 1) sum += 2;
     else{}
}
```

可以改成如下形式：

```
static const  struct PATTERN  patterns[PATTERN_NUM] ={12,2};
for(;;)
{ sum += patterns[mode].cnt; }
```

类似的例子：

```
if (b) a = c;
else a = d;
```

这段代码可以改写为：

```
static const type lookup_table[] = { d, c };
a = lookup_table[b];  //b 或者是 0，或者是 1。
```

如果更复杂一点：

```
if (b1) a = c;
else if (b2) a = d;
else if (b3) a = e;
else a = f;
```

改写为：

```
static const type lookup_table[] = { f, e, d, d, c, c, c, c };
a = lookup_table [b1 * 4 + b2 * 2 + b3];
```

（2）循环展开，最小值优化。

```
int length = array.size;
int min;
for(int i = 0 ;i<length;i++)
{
          if(array[i]<min)
                   min = array[i];
}
```

可以改写成：

```
#define DO1(x)  x
#define DO4(x)  x  x  x  x
#define DO8(x)  DO4(x)  DO4(x)
int length = array.size;
array.add_dummy_element(length%8);//增加 dummy 元素补齐到 8 的倍数
int min;
for(int i = 0 ;i < length;)
{
        DO8(min = ((array[i]+min) - abs(array[i] - min))/2;i++;)
}
```

绝对值函数编译器会给予特别优化，内部不存在分支判断，这样一来不仅降低了 i < length 的数量，而且去掉了一次循环内的比较操作，当然去掉分支指令也是有代价的，有时去掉后性能反而降低，需要在实践中不断提高判断的能力。

（3）告诉编译器哪个分支执行的可能性更大。

使用__builtin_expect 宏，详细可参考相关手册，本书不做展开。

最后，用目前比较流行的索引压缩算法——PForDelta 算法来考察实践环境中的用法。首先来认识一下这个算法。索引一章提到了在压缩过程中，文档间距（gap)越小，压缩的效果越好，有研究表明，在索引压缩的过程中 gap = 1 的情况接近 10%。假定一个索引块为 8 个 gap，并且从统计角度看，90%的情况下 gap 值小于 32，小于 32 的值均可以用一个 b=5bit 的数来表示。如图 8-3 所示。

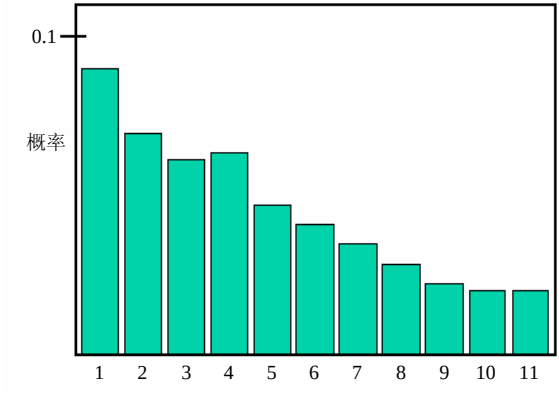


图 8-3　文档 Gap 分布规律

下面，建立这样一个结构 8*b-bit 的常规部分，看做是一个位数组，每个元素占 b-bit 定长空间，余下的为异常部分，看做是一个整形数组，每个元素占 4 字节定长空间。

假定有这样一个 doc 序列：23, 41, 8, 12, 30, 68, 18, 45（这是一个已经求过差分后的序列)。通过 PFordelta 方法的构造得到如下压缩结构，如图 8-4 所示。

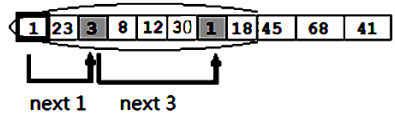


图 8-4　异常位置链表描述方式

椭圆框所示的部分为常规部分，常规部分的第一个值 1，表示从该地址开始，跳过 1 个地址，就可以找到下一个异常值的位置，同理第二个值 3 表示，跳过 3 个

地址，就是下一个异常值的位置。常规值从前到后存储，异常值从后向前存储。用二进制形式存储形式，如图 8-5 所示。

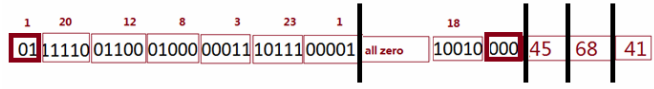


图 8-5　压缩结构的二进制存储形式

竖线隔开的是 5 个整型，前 2 个存储常规部分，后 3 个存储异常部分。值得注意的是跨块的问题，其中第 7 个位置上的 1 跨了 32bit 整形，因此一部分在第一个整形，另一部分在第二个整形，同时注意 bit 写入的时候，从低位往高位顺序写入，最后为了保证异常部分的地址能够按整型地址取得，对常规部分的剩余部分补零，实际实现时，可以假定不会出现需要补零这种情况，避免过多 if-else 分支。

这里只了解一下压缩后的形式，不讨论如何压缩的问题，仅就解压来考察如何将代码改为流水线友好的代码。

首先，假定将 8 个 b-bit 构成的 2 个整数（常规部分）和 3 个整数（异常部分）解压到 11 个整型的数组 block 中，依次为[1,23,3,8,12,30,1,18,45,68,41]，我们的目标是将该序列还原为原始序列[23, 41, 8, 12, 30, 68, 18, 45]。在对这个结构进行解码时，很容易想到需要对常规部分和异常部分做不同的解码操作，如下：

```
exception_pos = block[0]+1;
for(size_t i = 0,j = size_of_block;i < decompressed_num;i++){
    if(i != exception_pos){
              result[i] = block[i+1];}
    else{
              result[i] = block[--j];
              exception_pos += block[i+1] + 1;}
}
```

在循环中出现 if 判断，两个分支指令会影响流水线的流畅性，如何修改才能去掉这个 if 语句呢？可以采用两次循环的方法来处理。

```
for(size_t i = 0;i < decompressed_num;i++){
         { result[i] = block[i+1];}
}
exception_pos = block[0] + 1;
for(size_t i = exception_pos,j = size_of_block;i < size_of_block;i += block[i]+1){
```

```
        {result[i] = block[--j]; }
}
```

虽然做了两次处理，但由于充分利用了流水线的特性，代码执行速度大大加快。当然 PFordelta 算法并不是这样简单，只是出于举例简化的需要，感兴趣的读者可以参考笔者博客中编写的一段 PForDelta 可运行的代码[PForDelta code]。

简言之，流水线通畅的代码是那些多数为顺序执行的代码，较少或者没有跳转的代码，数据依赖较少的代码。并且，数据都是能在寄存器或者芯片缓存中获取的，否则都可能导致流水线的迟滞，影响效率。相关问题可以进一步阅读参考文献[Pipelining] ，[Compiler optimization]，[Code optimization]。

8.4　涉及内存的优化方法

8.4.1　概述

内存是一种重要的存储器，而存储器一般可以从速度、容量和价格等 3 个方面去考虑。内存恰恰是一个承上启下的作用，弥补了寄存器、cache 容量低的问题，同时弥补了磁盘访问速度低的不足。

内存在系统中的位置如图 8-6 所示[RE Bryant 2001]，内存通过系统总线和 CPU 相连，最昂贵的是寄存器，其次是 L1 cache,L2 cache，内存上的数据和代码都需要转化成更高级的存储形式才能做到可计算。

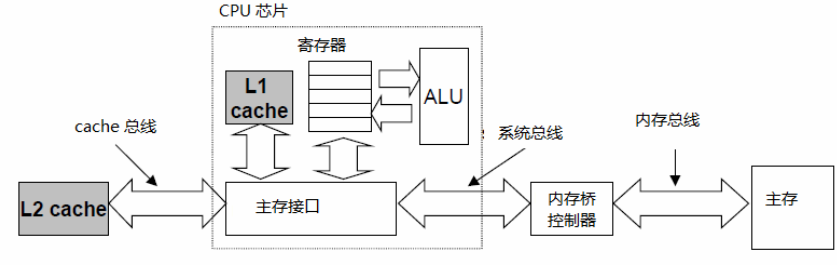


图 8-6　内存位置关系图

如图 8-7 所示，内存可以看做是多个内存结构组成[RE Bryant 2001]，而每个结构只承担部分的数据，在一次地址取值的过程中，由每个结构上的 supercell 来分别确定取值，这样的并行电路可以大大提高内存的带宽。

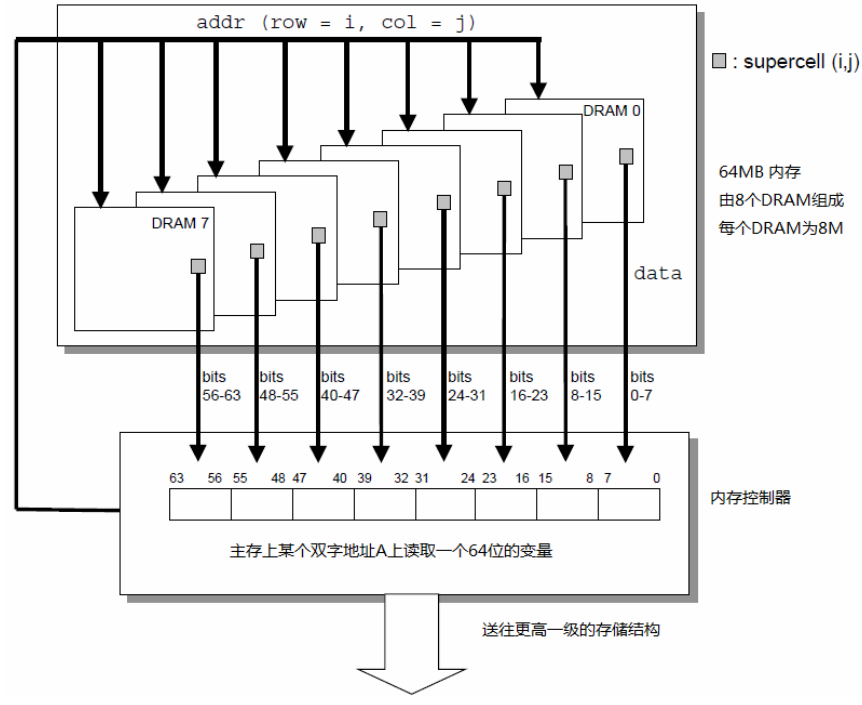


图 8-7　内存读取方式

相比磁盘，内存的发展非常迅速，但和 CPU 的发展相比尚有距离。在实际开发中，内存是最常接触到的存储介质，围绕内存的优化也最多，最复杂。

8.4.2　对换区

Linux 系统有一块特殊的设备介质叫做对换区（swap device），当物理内存不足或系统需要额外内存时，Linux 会按照一定替换算法，将若干当前使用频率低的内存页暂存在对换区中。对换区可以是一个磁盘，也可以是一个文件，最好是一个独立磁盘。

一般在大量需要内存的情况下，很容易出现对换区被写满，而且这种情况也很难避免，这时需要增加对换区的数量和容量。

由于交换区引发了一个问题称为 thrashing，通常发生在几个运行的进程同时需要很多内存，且需求量又大于物理内存的情况下。这时系统必须进行页面的换进换出来切换上下文环境，这就意味着大量的时间耗费在页面的换进换出，而不是执行代码上，结果使系统变得缓慢，CPU 时间被耗费在磁盘交换区上数据的换进换出上了。

对于一些特别重要的数据，在有把握的情况下（不存在内存泄漏），可以采用 mlock, mlockall 方法进行锁住，内存页一旦调入则不会被换回对换区。

内存被锁住而不交换到磁盘上，可用如下调用：r = mlock(ptr,size).

另一个锁定页面的函数：mlockall，它可以用 MCL_CURRENT 来锁定这个进程已经分配的所有页和用 MCL_FUTURE 来锁定这个进程将来要分配的所有页，以确保在系统资源接近耗尽依然可以使用类似 r = mlockall(MCL_CURRENT | MCL_FUTURE)这样的代码。

我们用一个小实验来考察下 mlock 的功效：

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <errno.h>
#if defined(__i386__)

static __inline__ unsigned long long get_cpu_ticks(void)
{
  unsigned long long int x;
  __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
                //.byte 0x0f,0x31 等价于 rdtsc，是另一种取机器码的方式
  return x;
                //改成__asm__ volatile ("rdtsc" : "=A" (x));效果一样
}               //关于操作码可以 intel 相关手册,或者[intel opcode]
#elif defined(__x86_64__)
static __inline__ unsigned long long get_cpu_ticks(void)
{
  unsigned hi, lo;
  __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
  return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}
```

```
#endif

#define LINESIZE 64
#define DATASIZE 1024*1024*64*4
#define LINECOUNT  DATASIZE/LINESIZE
char clear[DATASIZE];
void clear_cache()
{
         register char *p = (char*)clear;
         for(int i = 0;i< LINECOUNT;++i){
                    (*p)++; //每条 cache line 做一次写操，因为 clear 数
                            组足够大，写完后高级别 Cache 的内容替换为
                            clear 数组的内容
                    p+=LINESIZE;
         }
}
void visit_all(void* data)
{
                char *p = (char*)data;
                for(int i = 0;i< LINECOUNT ;++i){
                (*p)++;       //同 clear_cache，该操作相当于将 data 的数据
                              尽可能换入高级别内存中（L1 cache,L2 cache）。
                p+=LINESIZE;
                }
}

void adv(void* data)
{
          int ret = madvise(data,DATASIZE,MADV_DONTNEED);
        //madvise 向操作系统建议该段内存最近不再需要，建议对换
          printf("%d\n",ret);
}

void loc(void* data)
{
          int ret = mlock(data,DATASIZE);
          printf("%d\n",ret);
}
int main(int argc, char **argv)
{
          void* data1 = NULL;
          void* data2 = NULL;
          data1 =mmap(0,DATASIZE,PROT_READ|PROT_WRITE,MAP_PRIVATE|
          MAP_ANONYMOUS,0,0);         //mmap 匿名方式分配一块内存
          data2 =mmap(0,DATASIZE,PROT_READ|PROT_WRITE,MAP_PRIVATE|
```

```
    MAP_ANONYMOUS,0,0);          //
    visit_all(data1);            //将 data1 所指的虚地址，全部调页。
    visit_all(data2);
    const float CPU_MHZ = 3000.164;
  //用命令 cat /proc/cpuinfo 得到该值，代码中值为笔者测试机的测量值
    const float CPU_tick_count_per_second = CPU_MHZ*1000*1000;
    while(getchar()!='y')    //程序此时会顿住等待命令 y，用 top 命令
                             可以查看，当前程序占用内存为 512 兆。
    {          //读者可以自行试验如果没有调用 visit_all，当前程序占用
               内存非常少，因为没有访问，所以不调物理内存页
             printf("input command y\n");
    }
    adv(data1);              //建议操作系统将 data1 所指内存对换掉
    while(getchar()!='c')    //程序此时会顿住等待命令 c，用 top 命令可
                             以查看，当前程序中用内存为 256 兆
    {
             printf("input command c\n");
    }
    #ifdef LOC
    loc(data2);              //将 data2 锁定在内存中
    #endif
    #ifdef NOLOC
    adv(data2);              //将 data2 对换到对换区中
    #endif

    clear_cache();           //将 data2 从 L1，L2 cache 中赶走
    register int start, end;
    start = get_cpu_ticks();
    visit_all(data2);
    end = get_cpu_ticks();
    printf("in memory time  %d\n",(end-start)/ CPU_tick_count
              _per_second);//对在内存中的操作进行计时

    clear_cache();
    start = get_cpu_ticks();
    visit_all(data1);
    end = get_cpu_ticks();
    printf("out memory time %d\n",(end-start)/ CPU_tick_count
              _per_second);//对在对换区的操作进行计时

    munmap(data1,DATASIZE);
    munmap(data2,DATASIZE);
}
```

用如下命令进行编译：

```
g++ -g test.cpp -o test_loc -D LOC
g++ -g test.cpp -o test_noloc -D NOLOC
```

在笔者的试验机器上执行的结果如表 8-4 所示，可以看出在锁定内存后，效果提升十分明显，节约了调页的开销，在实时性要求高、同步性要求高、且内存用量不大的场合下，可以使用 mlock 锁定那些关键的数据，避免被对换。

表 8-4　mlock 实验数据

程　　序	锁定内存的效果（I）	对换在对换区的效果（O）	效果提升（O/I-1）
test_loc	0.147s	0.390s	165%
test_noloc	0.381	0.390	0.02%

笔者试验机的一些基本信息如表 8-5 所示，方便读者对比数据：

表 8-5　试验机基本数据

参　　数	值	获取方法
Linux 版本	Linux 67.23 2.6.18-92.el5	uname -a
CPU 主频	Intel(R) Xeon(TM) CPU 3.00GHz	cat /proc/cpuinfo
CPU 内核	4 核	cat /proc/cpuinfo
L1 cache size	1024	cat /proc/cpuinfo
L1 cache line size	64	getconf LEVEL1_DCACHE_LINESIZE
内存	4G	free -m
对换区	4G	free -m

8.4.3　cache line

CPU 通过芯片缓存(L1 Cache)和内存进行数据交互。交互的单位叫做 cache line，大小为 2 的幂，32 或 64 字节，虚拟内存页面大小为 4KB。cache line 的交互分为回填（refill）和回写（write-back）两种方式。

假定 CPU 需要从虚拟内存读取一个字节的操作数，其地址为 0xFFFFFFA3，cache line 的大小为 32 字节，则 CPU 需要将地址 0xFFFFFFA0 开始的 32 个字节全部读入，填充出一个完整的 cache line 后，然后从该 cache line 的第 4 个字节处取得该字节的内容。如果后继的指令也需要同样从 cache 行中读，那么填充 cache line 是值得的；否则，额外的填充 cache line 的时间就是浪费。因此指令和数据的局部性

越好，越符合 cache line 的设计要求。

回写的过程会引发内存一致性问题，当 A 处理器修改了 cache line 的内容，而 cache line 等价的内存区如果被 B 处理器使用了，B 处理器会得到一个错误值，从而等待 A 处理器将 cache line 数据写回，但遗憾的是如果 A 处理器跑的进程改写的是 cache line 前部的数据，而 B 处理器需要读的是后部的数据，因为这个原因，会产生看上去不必要的写回，这样同时也会导致流水线因为等待数据而产生停滞，下一节 fasle sharing 中将详细讨论。

为什么要 refill 呢？假定内存中一个 32 字节的连续地址，恰好是一条 cache line，程序要求在第一个字节的值清 0，由于 cache line 是整条整条进行数据交互的，如果不 refill，那么无法知道另外 31 个字节赋为何值，也就无法写入。

在一些特定场合里，例如需要对一片数据进行写入的时候，该片内存原数据全部作废不要，但是因为需要 refill，也需要将作废的数据读入 cache line，然后进行改写，之后才能写入。显然回填的操作是不必的，这时可以利用简短的汇编指令来避免这种回填的操作。这种技术叫做 non-temporal，其基本思想是当内存中的一块数据无用时，就不必对其进行 cache 的回填，non-temporal 的写操作不会回填 cache line，而是直接将新数据写入内存。涉及的指令包括 movntq，sfence。一个快速复制内存页的代码如下，cache line 为 64 字节（用命令 getconf LEVEL1_DCACHE_ LINESIZE 得到），下面是一个完整的例子：

```
#include <stdlib.h>
#include <string.h>
#include<stdio.h>
static void fast_copy_page(void *to, void *from,size_t page_cnt)
//以 4KB-Page 为单位进行复制，并默认 to，from 已对齐
{
          int i;
          unsigned char* t = (unsigned char*)to;
          unsigned char* f = (unsigned char*)from;

          __asm__ __volatile__ (
    //因为 asm，volatile 是保留字，因此需要用__asm__形式进行区别
                        "1: prefetchnta (%0)\n"
    //这些语句相当于将 from 指向的内存的一部分预取到 cache 中。
                        "   prefetchnta 64(%0)\n"
    // Prefectchnta 表示带有 Non-temporal 的功能，a 表示 all
                        "   prefetchnta 128(%0)\n"
```

```
                        "   prefetchnta 192(%0)\n"
                        "   prefetchnta 256(%0)\n"
                        "   prefetchnta 320(%0)\n"
                        "   prefetchnta 384(%0)\n"
                        "   prefetchnta 448(%0)\n"
                        "   prefetchnta 512(%0)\n"
                        "   prefetchnta 576(%0)\n"
                        : : "r" (f) );
      for(i=0; i<page_cnt*32; i++)
//每次执行复制 128 个字节，32 次循环，合计 4K 字节。
   {
          __asm__  (
                        "1: prefetchnta 640(%0)\n"
//这里 640 常量是因为 from 在汇编代码外面，每次循环已经加了一个 128 的偏移
                        "   prefetchnta 704(%0)\n"
//相当于一个移动的窗口，请读者画图即可明白。
                        "2: movdqa   (%0), %%xmm0\n"
                        "   movdqa 16(%0), %%xmm1\n"
                        "   movdqa 32(%0), %%xmm2\n"
                        "   movdqa 48(%0), %%xmm3\n"
                        "   movdqa 64(%0), %%xmm4\n"
                        "   movdqa 80(%0), %%xmm5\n"
                        "   movdqa 96(%0), %%xmm6\n"
                        "   movdqa 112(%0), %%xmm7\n"
                        #ifdef WITHOUT_NT
                        "   movdqa %%xmm0,   (%1)\n"
                        "   movdqa %%xmm1, 16(%1)\n"
                        "   movdqa %%xmm2, 32(%1)\n"
                        "   movdqa %%xmm3, 48(%1)\n"
                        "   movdqa %%xmm4, 64(%1)\n"
                        "   movdqa %%xmm5, 80(%1)\n"
                        "   movdqa %%xmm6, 96(%1)\n"
                        "   movdqa %%xmm7, 112(%1)\n"
                        #endif
                        #ifdef WITH_NT
                        "   movntdq %%xmm0,   (%1)\n"
                        "   movntdq %%xmm1, 16(%1)\n"
                        "   movntdq %%xmm2, 32(%1)\n"
                        "   movntdq %%xmm3, 48(%1)\n"
                        "   movntdq %%xmm4, 64(%1)\n"
                        "   movntdq %%xmm5, 80(%1)\n"
                        "   movntdq %%xmm6, 96(%1)\n"
```

```
                        "   movntdq %%xmm7, 112(%1)\n"
                        #endif
                        :: "r" (f), "r" (t) : "memory");
            f+=128; t+=128;
        }
        __asm__ __volatile__(
                        "sfence \n"
                        : :
                        );
};
int main(int argc,char**argv)
{
          const size_t page_count = atoi(argv[1]);
          const size_t s = page_count*(4*1024);
                                    //将 page_count 换上成字节数
          void* p = malloc(s); void* q = malloc(s);memset(p,0x1,s);
          for(int i=0;i<100;++i) {
          #ifdef FAST_COPY
               fast_copy_page(q,p,page_count);
          #endif
          #ifdef MEM_COPY
               memcpy(q,p,s);
          #endif
          }
          size_t sum = 0;
          unsigned char* r = (unsigned char*)q;
          for(int i=0;i<s;++i) {
                    sum += r[i];
          }
          printf("sum:%d\n",sum);                  //用以验证正确性。
          free(p); free(q); return 0;
}
```

分别用以下命令编译出 3 个程序。

```
g++ -g test.cpp -o test_memcopy -D MEM_COPY
g++ -g test.cpp -o test_withnt -D FAST_COPY -D WITH_NT
g++ -g test.cpp -o test_withoutnt -D FAST_COPY -D WITHOUT_NT
```

对多种 page count 进行试验，在笔者试验机上得到以下数据，如表 8-6 所示。

表 8-6　non-temporal 方法实验结果

page_count	memcopy(s)	withnt(s)	withoutnt(s)
1	0.004	0.003	0.002
100	0.015	0.019	0.013
200	0.075	0.046	0.056
300	0.13	0.071	0.115
5000	1.968	1.268	1.972
10000	3.9	2.588	3.71
15000	5.574	3.688	5.603
20000	7.44	5.121	7.391
25000	9.245	6.544	9.438
30000	11.293	7.533	11.058
35000	12.94	8.728	12.999
40000	14.799	10.148	14.973
45000	16.728	11.435	16.714
50000	18.576	12.862	18.285

可以看到在 page count 较小的情况下，没有使用 non-temporal 反而更好，在 page count 变大的情况下，带有 non-temporal 的快速复制速度快，而且耗费时间的增长趋势均为线性，如图 8-8 所示。

有兴趣的读者可以在 C 编程环境下编译运行这段程序，并请读者试图创造出更优化的方法，每一步执行的过程在其后均做了简要说明，建议读者对 movntq 和 sfence 指令进一步了解。

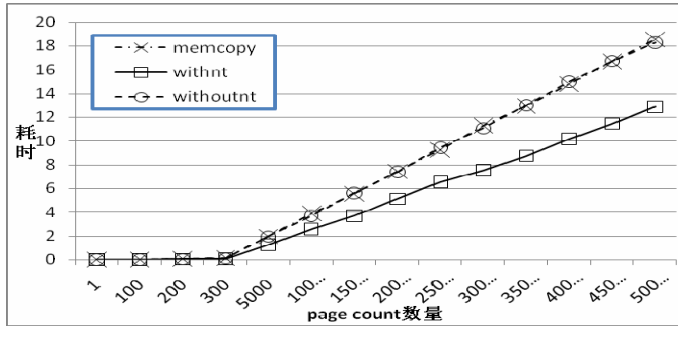


图 8-8　在使用 non-temporal 方法后改写的 memcpy 效果

对汇编不熟悉的读者可以参考下面 gcc 提供的函数，获得同样的功能。

```
#include <xmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);
void _mm_sfence(void);
```

8.4.4　false sharing 问题

在多处理器，多线程情况下，如果两个线程分别运行在不同的 CPU 上，而其中某个线程修改了 cache line 中的元素，由于 cache 一致性的原因，另一个线程的 cache line 被宣告无效，在下一次访问时会出现一次 cache line miss，哪怕该线程根本无须访问这个改动的元素。因为 False Sharing 问题，为了维护一致性，cache line 和主存来回写入导致系统性能降低，由于高层 cache 可看做是 CPU 一个部分，因此放在 CPU 中论述。

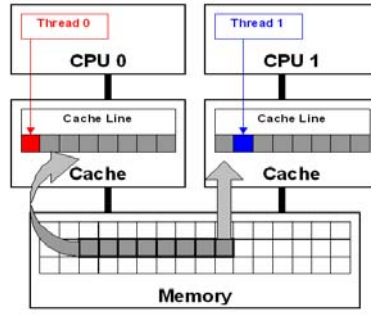


图 8-9　Falsh sharing 问题示意图

如图 8-9 所示，thread1 修改了 memory 灰化区域的第[2]个元素，而 Thread0 只需要读取灰化区域的第[1]个元素，由于这段 memory 被载入了各自 CPU 的硬件 cache 中，虽然从 memory 的角度看，这两种的访问时隔离的，但由于被紧凑地放在了一起，而导致了 thread1 的修改，在 cache 一致性的要求下，宣告了运行 Thread0 的 CPU0 的 cache line 非法，从而出现了一次 miss，导致从 memory 中重读到 cache line 的废操作，这个结果显然不是 thread0 所期望的。因此 SMP 的结构下，要特别注意 False Sharing 问题。[intel false sharing]给出了一般的解决方法，对于上例的两个线程的例子，并假定 cache line size = 64 字节，将线程两个全局变量均做 64 字节地址的对齐，即地址的后 6 位为 0 这样的地址上，例如 0xFFFFFFC0 这样的地址，可确保这两个变量不共存于一条 cache line 中。

```
size_t __attribute__((aligned(64))) thread0_global_variable;
size_t __attribute__((aligned(64))) thread1_global_variable;
```

或者采用对线程共享结构 padding 的方式，并对其在 64 字节上。

```
struct ThreadParams
{
   unsigned long thread_id;
   unsigned long v; //被用作读写的变量
   unsigned long start;
   unsigned long end;
   int padding[12]; //4 个无符号长整形+12 个整形 padding，恰好是 64 个字节
};
struct ThreadParams __attribute__((aligned(64))) Array[10];
```

为了表明 FALSE SHARE 带来的影响，设计了一个简单的多线程程序，包含两个线程，他们分别做求和使用不同的变量，但由于 cnt_1 的地址和 cnt_2 的地址在同一条 cache line 中，实测环境中 cnt_1 的地址为 0x600c00，cnt_2 的地址为 0x600c08，而 cache line 的大小为 64 个字节，这样就会发生 FALSE SHARING 问题。将两个变量在 64 字节对齐后，cnt_1 的地址为 0x600c40，cnt_2 的地址为 0x600c80，恰好错开在两条 cache line 上，源代码如下：

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#ifdef FS
size_t cnt_1;
size_t cnt_2;
#endif
#ifdef NONFS
size_t __attribute__((aligned(64))) cnt_1;
size_t __attribute__((aligned(64))) cnt_2;
#endif
void* sum1(void*)
{
        for(int i=0;i < 10000000;++i) {
                cnt_1 += 1;
        }
};
void* sum2(void*)
{
        for(int i=0;i < 10000000;++i) {
                cnt_2 += 1;
        }
```

```
};
int main()
{
        pthread_t*thread=(pthread_t*)malloc(2*sizeof(pthread_t));
        pthread_create(&thread[0],NULL,sum1,NULL);
                                                    //创建 2 个线程分别求和
        pthread_create(&thread[1],NULL,sum2,NULL);
        pthread_join(thread[0],NULL);
                                                    //等待 2 个线程结束计算。
        pthread_join(thread[1],NULL);
        free(thread);
        printf("cnt_1:%d,cnt_2:%d",cnt_1,cnt_2);
}
```

编译方法：

```
g++ fs.cpp -o test_nfs -g -D NONFS –lpthread
g++ fs.cpp -o test_fs -g -D FS –lpthread
```

用 time 命令计算耗时：time ./test_nfs，time ./test_fs。实测环境，test_nfs 耗时 0.037s，0.127s，在去掉 FALSE SHARING 后提速 243%，效果十分明显。

8.4.5　内存的锁问题

该内容涉及线程同步问题，放在本节是希望讨论关于 lock-free buffer 的基本想法。在多核编程中，由于指令在不同的 CPU 核上运行，因此内存同步问题尤为重要，需要用锁机制来控制访问顺序，而代价就是出现锁争用（lock contention）问题。

从一个例子开始讨论 lock contention 的几种常见解决办法。假如有一个链表，链表被多个线程共享，存在插入、查询和删除 3 个操作。为了避免访问冲突，很容易想到用一个锁，来同步整个表的访问。如表 8-7 所示。

表 8-7　链表的基本同步操作

删除操作	查询操作	插入操作
remove(item)	get(item)	insert(item)
{	{	{
ListLock.lock();	ListLock.lock();	ListLock.lock();
do_remove_item(item);	do_get_item(item);	do_insert_item(item);
ListLock.unlock();	ListLock.unlock();	ListLock.unlock();
}	}	}

很显然这样做锁空间（lock space）太大，没有必要被锁的代码也被纳入进去，并发性能差。是否可以降低锁空间，仅仅是对那些修改的节点进行加锁呢？对于 remove 操作，修改的节点为前驱节点的后继指针，如果采用这种方式，对于下面一个执行顺序，A：remove(a),B:remove(b)，指令 A 锁住了 head，B 锁住了 a，结果是只有 a 被删除了，而 b 没有被删除，如图 8-10 所示。

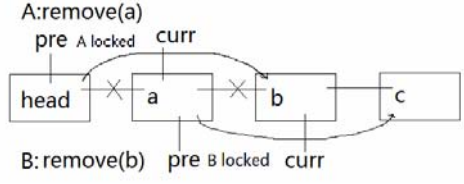


图 8-10　只对修改节点加锁的错误结果

这样的锁粒度虽然小，但导致了同步的错误。采用手递手的锁策略（Hand-over-Hand lock）可以解决这个问题。在删除的过程中，查找删除对象时，获得 pre 锁，再获得 curr 锁，依次向下，直到出现待删除的节点。读者可以自行画图证明这种方法的正确性，伪码如下：

```
Remove(item)
{
      Node pre = head; curr = null;
      int key = item.key;
      pre.lock();
      curr = pre.next;
      curr.lock();
      while(curr.key<key)
      {
          pre.unlock();
          pre = curr;
          curr = curr.next;
          curr.lock();
      }
      If(curr.key == key)
      {
          pre.next = curr.next;
          return true;
      }
          return false;
      }
      curr.unlcok();
```

```
        pre.unlock();
    }
```

前面讨论了降低锁空间的方法，但锁的机制本质上是让等待锁的线程进入阻塞态，从而等待唤醒，这样总是存在上下文切换的代价。是否可以利用多核的环境，在锁空间很小的情况下，持续等待直到竞争条件解除呢？下面即将介绍的 lock-free 方法正是这样一种思路，目前 lock-free 的方法大多采用了 CAS 和 FETCHANDADD 这两个基本原子操作，通过一段代码来有个感性的认识：

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <syscall.h>
#if defined(__x86_64__)
          #define ATOMICOPS_WORD_SUFFIX "q"
                              //64 位环境下使用 cmpxchgq 命令
#else
          #define ATOMICOPS_WORD_SUFFIX "l"
                              //32 位环境下使用 cmpxchgl 命令
#endif
static inline bool compare_and_swap(volatile size_t *p, size_t val_old,
size_t val_new){
           char ret;
           __asm__ __volatile__("lock; cmpxchg" ATOMICOPS_WORD_
           SUFFIX " %3, %0; setz %1"
                              //lock 命令锁总线，因此可以保证多核同步
                               : "=m"(*p), "=q"(ret)
                              //setz 为 ZF 符号位是否置位，用于设置返回值
                               : "m"(*p), "r" (val_new), "a"(val_old)
                               : "memory");
           return (bool)ret;
}
static inline size_t fetch_and_add(volatile size_t *p, size_t add){
           unsigned int ret;
           __asm__ __volatile__("lock; xaddl %0, %1"
                              :"=r" (ret), "=m" (*p)
                              : "0" (add), "m" (*p)
                              : "memory");
           return ret;
};
```

```
struct my_cas
{
        my_cas(unsigned char t):m_val_old(t){}
        size_t m_val_old;
        inline void try_continue(size_t val_old,size_t val_new){
                while(!compare_and_swap(&m_val_old,val_old,val
                _new)){};
        }
        inline void add(size_t val_new){
                fetch_and_add(&m_val_old,val_new);
        }
};
volatile size_t g_uCount;
pthread_mutex_t g_tLck=PTHREAD_MUTEX_INITIALIZER;
my_cas mutex(1);
const size_t cnt_num = 10000000;
void* sum_with_mutex_lock(void*)
{
        for(int i=0;i < cnt_num;++i) {
                pthread_mutex_lock(&g_tLck);
                g_uCount += 1;
                pthread_mutex_unlock(&g_tLck);
        }
};
void* sum_with_f_and_a(void*) //用 fetch_and_add 原子操作来保证结果正确性。
{
        for(int i=0;i < cnt_num;++i) {
                fetch_and_add(&g_uCount,1);
        }
};
void* sum_with_cas(void*)    //用 CAS 原子操作来模拟锁操作。
{
        for(int i=0;i< cnt_num;++i)
        {
                mutex.try_continue(1,0);
                g_uCount += 1;
                mutex.try_continue(0,1);
        }
}
void* sum_with_cas_imp(void*)
{
   for(int i=0;i< cnt_num;++i) {
      for(;;) {
```

```
            size_t u = g_uCount;
            if(compare_and_swap(&g_uCount,u, u+1)){//在上一条语句和本条语句
之间，g_uCount 无篡改则进行加 1，
                break;//break 出该循环，否则重试，直到成功。
            }
        }
    }
}
void* sum_with_cas_imp_yield(void*)
{
    for(int i = 0;i < cnt_num;i++) {
        for(;;){
            size_t n = 1;
            for(;n<4096;n<<=1){
                for(size_t i = 0;i < n;i++){
                __asm__ ("PAUSE");//PAUSE 可以看作一个小延迟函数，有利于
缓解 CPU 的 100%使用率状况。
                }
                size_t u = g_uCount;
                if(compare_and_swap(&g_uCount, u,u+1)){
                    break;
                }
            }
            if(n < 4096){ break; }


             else{
                syscall(SYS_sched_yield);
            }
        }
    }
}
void* sum_just_free(void*)
{
        for(int i = 0;i < cnt_num;++i) {
                g_uCount += 1; //完全无锁，无等待，但执行结果通常是错误的。
        }
}
void* sum(void*)
{
        #ifdef M_LOCK
              sum_with_mutex_lock(NULL);
        #endif
```

```
            #ifdef FETCH_AND_ADD
                  sum_with_f_and_a(NULL);
            #endif
            #ifdef FREE
                  sum_just_free(NULL);
            #endif
            #ifdef CAS
                  sum_with_cas(NULL);
            #endif
            #ifdef CAS_IMP
                  sum_with_cas_imp(NULL);
            #endif
            #ifdef CAS_IMP_YIELD
                  sum_with_cas_imp(NULL);
            #endif
};
int main()
{
            pthread_t* thread = (pthread_t*) malloc(10*sizeof( pthread_t));
            for(int i = 0;i < 10;++i){
                        pthread_create(&thread[i],NULL,sum,NULL);
            }
            for(int i = 0;i < 10;++i){
                        pthread_join(thread[i],NULL);
            }
            printf("g_uCount:%d\n",g_uCount);
}
```

用以下编译命令编译出 6 个程序

```
g++ test.cpp -o test_free -D FREE -lpthread
g++ test.cpp -o test_fetchandadd -D FETCH_AND_ADD -lpthread
g++ test.cpp -o test_mlock -D M_LOCK -lpthread
g++ test.cpp -o test_cas -D CAS -lpthread
g++ test.cpp -o test_cas_imp -D CAS_IMP –lpthread
g++ test.cpp –o test_cas_imp_yield –D CAS_IMP_YIELD -lpthread
```

分别用 time 命令执行，得到如下结果，如表 8-8 所示。

表 8-8　各种方法性能比较

源　程　序	real（实际执行时间）	user	sys
free	1.103s	4.050s	0.001s
pthread_lock	48.546s	44.975s	2m26s

续表

源　程　序	real（实际执行时间）	user	sys
fetchandadd	5.569s	21.820s	0.006s
cas	1m38s	6m29s	0.021s
cas_imp	22.095s	1m27.517s	0.010s
cas_imp_yield	18.866s	1m15.158s	0.019s

直观可以看出，cas 和 pthread_lock 同为锁的机制，效果最差。而 cas_imp、cas_imp_yield 的同步方式次之，fetchandadd 表现最好，接近无锁的形式，有兴趣的读者可以用 strace –c　test_mlock 这个命令考察 pthread_lock 系统调用开销大的原因，了解 futex 系统调用的机制，cas_imp 和 cas_imp_yield 在本例对内存的竞争上效果相当，但在竞争一些慢速资源例如网络数据，硬盘数据的时候，适当地进行 PAUSH 和 yield 系统调用是值得的。

表中的数据取多次实验数据中的某一次，虽每次测试都有不同，但对比关系基本保持一致，有兴趣的读者可以自行实验。

这里特别要强调 compare_and_swap 这个原子操作，它源于 IBM System 370，其包含 3 个参数：共享内存的地址（*p），该地址期望的值（old_value），一个新值（new_value）。只有当*p == old_value 时，才产生交换操作，返回真值，否则返回假值，相当于如下代码[Andrei 2007]：

```
template<class T>
bool CAS(T* addr, T exp, T val)
        //只有在整个函数过程具有原子性时才正确，实际的代码参照此前的事例代码。
{
       if（*addr == exp）{
       *addr = val;
       return true;
       }
       return false;
}
```

在研究 lock-free buffer 之前，有 4 点值得注意：

（1）什么是 lock-free 的代码？在对竞争资源的访问时，lock 和 unlock 成对出现的代码就是有锁的代码，同步依赖于加锁，开锁，而 lock-free 的代码不具有这种情况。

（2）lock-free 和 wait_free 的关系。lock-free 不等于 wait-free，是否需要 wait-free 很大程度受业务和策略决定，本书不展开 wait-free 的讨论。

（3）spining 和 blocking 的区别。spining 表示忙等待，被锁的时间短，该提法一般用于多核环境。blocking 是线程被挂起，OS 调度其他线程，等待时间一般较长。

（4）spin lock 和 lock-free 的关系。spin lock 是 Lock-free 的一种应用，spin lock 特指那些通过循环去等待（而不是用挂起然后等待唤醒的方式）资源释放的轻量级同步方法，spin lock 只在多核系统才有意义，借助 compare_and_swap 或 test_and_set 来实现。

论文[M. Michael 1996]中提到的两种并发队列的方法是这方面早期的研究，易于理解。首先来看双锁的并发队列：

```
structure node t    { value: data type,  next: pointer to node t}
structure queue t   {Head: pointer to node t, Tail: pointer to node t,
H_lock: lock type, T_lock: lock type}

initialize(Q: pointer to queue t)
node = new node()    # 分配自由节点,相当于队列中有一个伪节点(dummy object)
node->next = NULL                       #
Q->Head = Q->Tail = node                # 头尾节点均执行该自由节点
Q->H_lock = Q->T_lock = FREE            # 头锁和尾锁均打开

enqueue(Q: pointer to queue t, value: data type)
        node = new node()                       # 分配一个新节点
        node->value = value                     # 赋值操作
        node->next.ptr = NULL                   #
        lock(&Q->T lock)          # 入队只需要尾锁，因此只对尾锁加锁。
              Q->Tail->next = node              # 当前节点入队
              Q->Tail = node                    # 尾指针后移
        unlock(&Q->T lock)                      # 解锁

dequeue(Q: pointer to queue t, pvalue: pointer to data type): boolean
        lock(&Q->H lock)                      # 出队只需要头锁，对头锁加锁。
              node = Q->Head                    #
              real_head = node->next
    # 拿到头节点，Head 的 next 节点是自由节点，自由节点的 next 节点是头节点
        if new head == NULL                     # 如果是空队列
               unlock(&Q->H lock)               # 释放头锁并返回。
               eturn FALSE                      #
        endif
```

```
            *pvalue = real head->value      # 取出头节点值
            Q->Head = real head->next       # 队列中拿掉头节点
        unlock(&Q->H_lock)                  # 释放头锁
        free(real head)                     # 删除头节点
        return TRUE      # Queue was not empty, dequeue succeeded
```

这种头尾双锁的方式，使得入队和出队互不干涉，实现简单，易于理解。但问题是并发的入队和出队会出现锁争用的情况，论文[M. Michael 1996]继续提到了一种 lock-free 的并发队列访问方式。

```
structure pointer t      {ptr: pointer to node t, count: unsigned
integer} #结点结构包含数据结点和计数器
structure node t         {value: data type, next: pointer t}
structure queue t        {Head: pointer t, Tail: pointer t}

initialize(Q: pointer to queue t)
           node = new node()          #分配一个自由节点，相当于队列中有一
                                      个伪结点（dummy object）
           node->next = NULL          #初始化自由节点
           Q->Head = Q->Tail = node   #初始化头、尾节点

enqueue(Q: pointer to queue t, value: data type)
E1:      node = new node()            #分配一个节点
E2:      node->value = value          #节点赋值
E3:      node->next = NULL            #
E4:      loop         #一直尝试，直到入队成功，注意这里不是 wait-free 的代码。
E5:             tail = Q->Tail        #取下队尾
E6:             next = tail->next     #取出真队尾
E7:             if tail == Q->Tail    # Q->Tail 是否被其他线程篡改？如果
                                      篡改直接回到 loop
E8:                       if next == NULL
                                      # Tail 是否指向的是最后一个节点
E9:                           ifCAS(&tail->next,next,<node,next.
                          count+1>)
                              #尝试 tail->next = <node,next.count+1>
                              #成功则 break 返回，失败则继续
                              #循环找新队尾试图插入。
E10:                                 break
E11:                          endif
E12:             else
E13:                            CAS(&Q->Tail,tail,<next,tail.count+1>)
                              #尝试修改队尾，更新到最新队尾
E14:                      endif
```

```
E15:        endif
E16:    endloop
E17:    CAS(&Q->Tail, tail, <node, tail.count+1>)
                              #完成入队，入队后可能有两种可能。

dequeue(Q: pointer to queue t, pvalue: pointer to data type): boolean
D1:   loop                                          # 一直尝试，直到出队成功
D2:           head = Q->Head                        # 读出头节点
D3:           tail = Q->Tail                        # 读出尾节点
D4:           next = head->next                     # 读出第二头节点
D5:           if head == Q->Head                    # 头节点是否被篡改
D6:                   if head == tail               # 队空或者队尾落后？
D7:                           if next == NULL             # 队空？
D8:                                   return FALSE        # 队空，无法出队
D9:                           endif                       # 队尾落后
D10:                                    CAS(&Q->Tail, tail, <next,
                                  tail.count+1>) # 队尾指向第二头节点
D11:                  else
D12:                  *pvalue = next.ptr->value  # 取出第二头节点值
D13:                  if CAS(&Q->Head, head, <next, head.count+1>)
                                                    #头节点指向下一个节点
D14:                           break                # 完成出队，退出
D15:                  endif
D16:              endif
D17:          endif
D18:    endloop
D19:    free(head.ptr)                              # 释放出队节点。
D20:    return TRUE
```

对以上程序正确性的证明，本章不再展开，只是对 E13 这段代码做一个说明，这是理解这段代码的关键。这段代码执行 CAS 如果返回 ture，表示发生了交换，则队尾得到了跟进，如图 8-11（a）所示；如果返回 false，表示未发生交换，队尾出现了滞后，如图 8-11（b）所示。因为存在这种情况，在出队和入队的过程中都有机会做队尾的纠正工作，可以保证高效且正确。

写一个正确的 lock-free 的程序是比较复杂的，因为没有用到锁，会出现类似 D2 和 D5 这样的语句，多线程环境下，地址的变量在不同时间取得数据都会不同，因此程序需要考虑各种因素，CAS 的功效正是在于只有无篡改的情况下，才发生一次正确更新，且这次更新是一次原子操作，因此保住了程序的正确性。

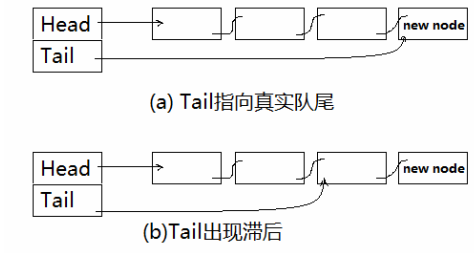


图 8-11　lock-free queue 代码中 Tail 可能的两种情况

8.4.6　内存库的使用

在多线程环境下，需要频繁分配和释放内存，使用 GLIBC 提供的 Malloc 和 Free 函数往往不够理想，这时可以考虑 SmartHeap、Hoard 和 Intel TBB 等内存库，本节仅对 Hoard 线程库的原理做简要说明，以考察这些内存库主要面向的问题和解决问题的原理[Berger 2000]。

在[Berge 2000]这篇论文中，归纳了 Hoard 面向的 4 个问题。

（1）速度（Speed）：在多核环境下，内存分配要足够快。

（2）扩展性（Scalability）：在处理器增加的时候，allocator 的性能必须线性可扩展，以确保应用可扩展。

（3）False sharing avoidance：避免在同一个处理器上的多个线程出现共享同一条 cache line 数据的情况。

（4）低碎片（Low Fragmentation）：碎片定性为从系统分配的全部内存除以系统实际要求的内存，过多的碎片可以导致数据局部性的降低，并导致额外的对换。在 hoard 系统，fragmentation 也叫 blowup，我们来考察这样的情况，假定一个进程有 P 个线程，每个线程执行 x=malloc(s);free(x);如果这些线程串行的执行，整个内存需求量为 s，但如果该进程是运行在 P 个处理器内核上，每个调用 malloc 可能是完全并行的，这就导致内存的需求量最大值为 $P*s$，而实际上因为多核还需要在这个耗费上乘以一个 P 的因子，因此内存的耗费增加到 P^2*s[Emery 2000]。Hoard 给出的解决方案可以在一定程度上解决这个问题。

Hoard 提出每处理器一个局部堆，并共享一个全局堆，即每个线程可以访问属于自己的内存堆和全局堆，但不能跨本地堆。Hoard 将 0 号堆定义为全局堆，在 P

个处理器核的系统环境下，定义 1—P 共计 P 个本地堆，为了避免并发的线程使用同一个堆，对本地堆进行了 double，即每个 1 个全局堆，2P 个本地堆。并给出这样的分配的策略：建立 thread_id 和 heap_id 的映射，某个线程只会在某个特定的堆上分配内存。当本地堆内存使用降低到某个阈值一下，则会切割一块内存给全局堆，以便于其他局部堆使用。

请注意两个细节：

（1）某个线程 ID 只会申请特定的本地堆，但却可以释放其他本地堆的内存。这很容易理解，线程之间会传递参数，可能一个线程申请的内存，最终由另一个线程释放（生产者消费者类程序）。

（2）假定有 4 个 CPU 核（0,1,2,3）、4 个线程（0,1,2,3）、4 个本地堆（0,1,23），映射关系是 *i* 号线程在 *i* 号堆上申请内存，但并不表示 *i* 号 CPU 只和 *i* 号堆交互，线程是可能在 CPU 核上切换的，在 A 时刻 *i* 号线程在 *i* 号 CPU 上，在 B 时刻 *i* 号线程可能在 *j* 号 CPU 上，因此对于某个 CPU 核来说 4 个本地堆都是可以访问到的，但在一个特定的时间内某个 CPU 上执行的线程 *i* 申请的内存都在堆 *i* 上，在这个特定时间内 CPU 对应了一个特定的堆。

[Berge 2000]给出了分配和释放内存的伪码，如下：

```
malloc (sz)
1. If sz>S/2, allocate the superblock from the OS and return it.
                                  //如果分配的内存过大采用 mmap 的方法进行分配
2. i ← hash(the current thread) .  //类似 thread_id%P 的哈希函数，P 为 CPU 数量。
3. Lock heap i.                   //对 Heapi 加锁
4. Scan heap i's list of superblocks from most full to least (for the
size class corresponding to sz). //在堆中查找刚刚好能满足内存需要的内存块
5. If there is no superblock with free space,     //若本地堆无法分配
6. Check heap0(the global heap) for a superblock.
                                  //检查全局堆是否有空闲内存块
7. If there is none,              //若全局堆也没有合适内存块
8.     Allocate S bytes as superblock s and set the owner to heap i.
                        //从操作系统分配一块超级快，并且将该块纳入堆 i 进行管理
9. Else,                                        //若全局堆有合适内存块
10.Transfer the superblock s to heap i.   //将该块整体切给堆 i 进行管理。
11. u_0 ← u_0 − s.u                       //并作相应数量关系赋值。
12. u_i ← u_i + s.u
13. a_0 ← a_0 − S
```

14. $a_i \leftarrow a_i + S$
15. $u_i \leftarrow u_i + sz$　　//堆 i 的使用内存数增加。
16. $s.u \leftarrow s.u + sz$　　//涉及的超级快内部的使用数增加。
17. Unlock heap i.
18. Return a block from the superblock.

free (ptr)
1. If the block is “large”,　//如果待释放的内存过大，采用 munmap 方式释放。
2. Free the superblock to the operating system and return.
3. Find the superblocks this block comes from and lock it.
4. Lock heap i, the superblock’s owner.
　　//将本地堆和涉及的超级块进行加锁操作，
5. Deallocate the block from the superblock.　　//进行释放操作
6. $u_i \leftarrow u_i - block\ size$
7. $s.u \leftarrow s.u - block\ size$
8. If i=0, unlock heap iand the superblock and return.
　　//如果释放的内存来自全局堆，直接返回。
9. If $u_i < a_i - K*s\ and\ u_i < (1-f)*a_i$, //当堆 i 的使用量极小，以至于满足了某些条件
10. Transfer a mostly-empty superblock s1 to heap 0 (the global heap).
　　//将堆 i 的多余超级块归还全局堆
11. $u_0 \leftarrow u_0 + s1.u;\ u_i \leftarrow u_i - s1.u$
12. $a_0 \leftarrow a_0 + S;\ a_i \leftarrow a_i - S$
13. Unlock heap i and the superblock.

同时，通过一个实际的例子，来了解这个过程，如图 8-12 所示。

假定有这样一个分配和释放的顺序，其中 t1 表示线程 1：

```
t1: x9 = malloc(s)
t1:free(y4);
t2:free(x2);
t2:free(x9);
```

可见在线程 t2 释放 x9 后，堆 1 的第一个超级块空闲了一半，而第二个超级块全空，这样满足了收缩（cascade）条件，因此将堆 1 的这个超级块归还了全局堆，以便于分配给其他本地堆。这其实是一种平滑使用需求的技巧，在搜索引擎建立索引也会用到，虽然按照文档均分到各个索引节点，但制作出的索引文件大小和最后进行查询的性能，总是出现参差不齐的情况，由于搜索的时间决定于最慢的索引节点，因此如果平滑没处理好，将会导致很大的问题。

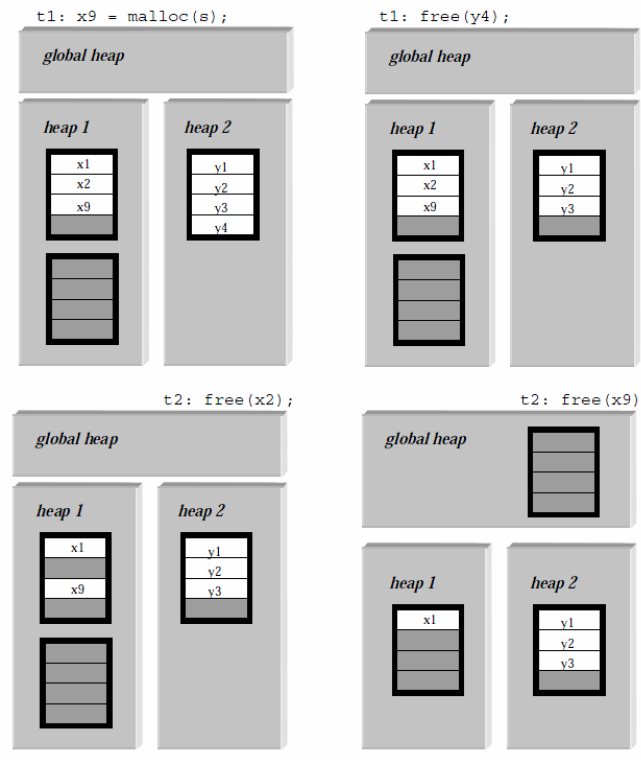


图 8-12　Hoard 内存分配和释放顺序示意图

这种设计为什么能解决以上提出的问题呢？

首先，Hoard 解决 Blowup 的问题正是通过本地堆和全局堆之间的空闲块互换实现的，论文[Emery 2000]从数学上给出了证明，由 Hoard 产生的碎片是有上界的，Hoard 的 Fragmentation（分配的内存数/使用的内存数）在 9 个经典实验程序中除了 shbench 为 3.17 其余均在 1.2 左右，也就是说，平均申请 1 份内存，只需搭 0.2 份碎片。

其次，Hoard 解决 FALSE SHARING 问题是通过多堆策略实现的。在多核环境下，不同的执行线程被区分在不同的堆中，任意两个 CPU 访问到同一个堆中内存的概率极低，在此前举得 FALSE SHARING 的例子中，两个线程分派到两个堆，使用的内存相隔甚远，因此出现 FALSE SHARING 的概率较低，论文[Emery 2000]还从 active 和 passive 两种 FALSE SHARING 情况进行了讨论，Hoard 在这两种情况都能很好地解决。

此外，Hoard 的多堆策略很好地解决了内存分配器锁争用的情况，在一个堆的传统分配策略中，虽然内存被集中管理，但集中管理带来了同步地开销，每次申请都需要加锁串行化申请，在多堆策略中就可以进行并行化的申请，彼此不影响，而同时两个线程被哈希到一个堆的概率，且同时需要争夺 Heap Lock 的概率极低。当然这是在线程自己申请内存，自己释放内存的情况下，如果程序中总是一个线程申请内存，而由其他线程释放该内存，则锁争用的情况依然不可以避免。生产者消费者问题的程序就属于这一类。一般这类程序在 Hoard 上运行得都不好。

最后，关于 Scability，我们希望 Hoard 能够在处理器核增加的同时，表现更佳，引用论文[Emery 2000]的一幅图来说明，如图 8-13 所示。

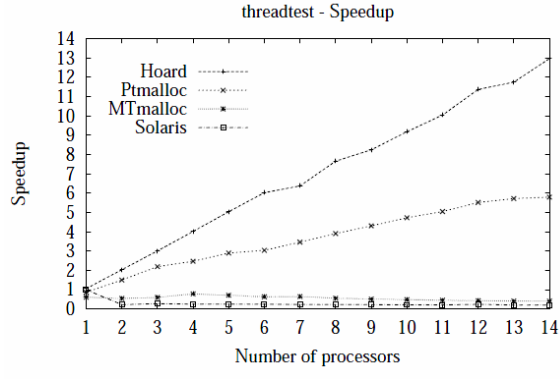


图 8-13　Hoard 在可扩展性上的表现

这是用 threadtest 测试程序测试的结果，纵坐标的刻度表示和 Solaris 单堆分配策略相比的加速比（Speedup），因为单堆策略基本上在 CPU 增加的同时，分配内存速度基本保持不变。Hoard 在 2 个 CPU 时 Speedup=2 表示，在同等情况下，Solaris 每秒分配 X 兆内存，而 Hoard 每秒分配 2X 兆内存。从趋势上看，Hoard 实现了 scability，14 核时加速比接近 14，并发的效果极好。

最后，我们对 Hoard 的理解可以从 SMP 架构出发，看做是将多个 CPU 共享的内存进行了一定规范化的划分，使内存出现了一些私有化的性质，但同时又能共享、共通。

通过对 Hoard 内存库的学习，综合此前介绍的知识，内存分配和使用是程序优化的重点，为了程序执行的局部性，可以把相关的数据集中处理，为了避免多线程的同步等问题，又需要按照一定的方法将内存使用区隔开来，同时还要注意这种区

隔导致的碎片，引起性能的参差不齐，需要通过一些方法来进行平滑。

8.5 涉及磁盘的优化方法

磁盘是计算机上最慢速的设备，但存储容量大，成本低，是很好的存储后备。近年来，受到 CPU 和内存的快速发展的刺激，固态磁盘的出现使得磁盘家族增加了新的研究热点，搜索引擎的很多模块也都开始换装固态磁盘。无论是机械磁盘还是固态磁盘，在我们做大规模日志分析和处理、海量数据挖掘等过程中都无法回避，对磁盘的调优显得尤为重要。本章主要关注机械磁盘的调优，不特别指出的均表明是机械磁盘。

磁盘慢到底慢在哪里？一次磁盘 IO 主要耗费的时间包括旋转时间、寻道时间和传送时间。旋转时间和寻道时间也称作寻址时间，磁盘和内存之差主要就在寻址时间上，内存是电路寻址，而磁盘是机械寻址。特别是寻道时间是慢中之慢，随机 IO 和顺序 IO 速度相差极大，就是最好的证明。哪些方法可以改善磁盘 IO 的性能呢？本节将从多个角度进行论述。

8.5.1 磁盘 IO 的调度

在讨论磁盘调度算法的选择之前，先了解 4 个概念。

（1）归并。在 IO 请求的队列中，如果当前的请求和下一个请求的磁盘地址是相邻或者邻近的时候，会执行一次归并操作，将这两个请求合并为一个请求，只发生一次寻道和读盘操作，从而减少 1 次 IO。

（2）排序。假定当前队列是一个按照磁盘物理地址有序的，IO 请求地址分别为：A，B(A+100),C(A+200),D(A+500)，一个新的请求 E，距离 A 的偏移为 300，则把 E 的请求插入到 C 和 D 之间，使得 IO 在物理地址上有序。因此排序并不是每来一个 IO 请求进行一次排序，而是做一次插入排序，排序的结果是可以使 IO 获得更多的归并机会，同时减少寻道的时间，让磁头尽可能不要来回滑动。

（3）饥饿。归并并不会产生饥饿，而排序会产生饥饿，可以想象如果一个读取磁盘较大地址的 IO，就可能产生饥饿，因此对于在队列中等待时间较长的请求应给予更高的优先级。

（4）依赖。通常读请求是有先后依赖关系的，而写请求常常可以是异步的，因此读请求更倾向于先进先出的调度算法，不会造成后读的完成后还有等待前面未读的 IO 请求的结果。写操作更倾向于批量地进行，不希望“碎写”，因此 IO 之间的依赖关系也是必须考虑的因素。

Linux 磁盘调度算法有如下 4 种[Linux Symposium 2005]：

（1）noop：只进行简单的 IO 归并，不执行 IO 排序或其他减少磁头寻道的操作，接近于先进先出，只有发生归并的时候，排在后面的会有提前的机会。

（2）deadline：主要是避免 Writes-starving-reads 问题，一般写是异步的，而读是同步的，如果大量写操作会导致读取受到影响，deadline 调度算法会给每个 IO 请求一个最迟响应时间。该调度算法含 2 个队列，读队列和写队列，同时支持归并和排序的功能。

（3）Anticipatory：继承了 deadline 的功能，同时增加了应对批量 IO 的一个功能，在顺序读取大文件的时候，该调度算法在得到读取 IO 后，等待一段时间（默认 6 毫秒），以预料（anticipate）其后可能的 IO，将这些 IO 归并在一起，统一排队。

（4）cfq(Complete Fair Queue)：完全公平的调度算法，进程内部先来先服务，例如进程 A 的 IO 请求在进程 A 的队列上上排队，进程 B 的 IO 请求在进程 B 的队列上排队。各进程队列间采用时间片（round robin）的方法进行磁盘的分配。

修改磁盘调度算法的方法如下：

```
cat /sys/block/hda/queue/scheduler              //查看系统当前调度算法
echo noop > /sys/block/hda/queue/scheduler      //修改为 noop 调度算法
```

修改后，在`/boot/grub/grub.conf` 文件中，增加一行 `elevator=noop` 重启操作系统即可。

注意：在选择了某种调度算法后，还可能存在一些后续的参数调优，例如 deadline 调度算法下，需要修改 read_expire 和 write_expire。这些请读者参考相关手册。调度算法选择的依据，也需分情况。

使用 noop 的情况：在使用固态磁盘，随机 IO 比顺序 IO 更多，对 CPU 开销敏感的情况下适合使用 noop 调度算法。noop 的主要优点是调度算法简单。使用 deadline 的情况、数据库这种算力强大，瓶颈在磁盘的系统。使用 anticipatory 的情况：数据主要呈现顺序读写的系统，需要极高的吞吐量的后台处理系统。使用 cfq 的情况：多用户，多处理器的系统，cfq 一般是默认配置，是在各种情况下表现最稳定的调

度算法。

最后，在实际操作中，我们不能总是依赖操作系统提供的排序，在应用程序上也可以做这种 IO 的排序操作，将随机的 IO 进行累积、排序后，再提交给操作系统，这样可以在更大程度上减少磁头来回移动的距离，因为操作系统排队很短，能够优化的空间只有短短的一个队列中的请求，因此在应用程序上优化可以获得更出色的性能提升。

另外，由于这种排队机制，在随机读的场合，多线程可以达到单线程的数倍，充分提高读取的效率，同时还可以使用异步 IO 的方式，把排序和归并的工作交给操作系统，获得更有的 IO 性能。

8.5.2 其他常见磁盘参数调优

在搜索引擎的实践中，顺序读的机会很多，例如从网页库中顺序读出网页做索引、索引合并等。大规模日志做后续分析工作，通常可以增加磁盘预读来作改进。

```
cat /sys/block/<disk_subsystem>/queue/read_ahead_kb    //读取预读数量
echo 4096 > /sys/block/<disk_subsystem>/queue/read_ahead_kb
                                                    //预读 4M 字节
```

磁盘 IO 请求队列 nr_request(kernel request queue)也是一个很重要的常见需要修改的，通常认为必须是 queue_depth (hardware requests queue)的两倍，该参数的提高有助于合并更多的读写。

```
cat /sys/block/<disk_subsystem>/queue/nr_requests  //查看系统当前设置
echo 128 >/sys/<disk_subsystem>/queue/nr_requests
                                                    //将等待队列长度设置为 28
```

在大规模写入的情况下，Linux 通常借助 pdflush 来进行批量写入磁盘，可以通过 cat /proc/sys/vm/nr_pdflush_threads 来查看 pdflush 当前线程数，当写入量巨大时，系统会自动孵化出新的线程参与到写磁盘工作，在写入量变小时，会自动减少线程数。

对 pdflush 的唤醒受以下 4 个参数的影响：

```
/proc/sys/vm/dirty_writeback_centisecs：默认 500，含义是每 5 秒唤醒一次
pdflush 线程，有多少唤醒多少。
/proc/sys/vm/dirty_expire_centisecs：默认 3000，l 含义是 inux 系统最坏情
```

况下 30 秒才进行一次写操作，即应用程序的 write 在最坏情况下，需要熬 30 秒，操作系统才会真正执行。
/proc/sys/vm/dirty_ratio，默认 40，脏页面可以占整个内存的最大比例。
/proc/sys/vm/dirty_background_ratio：默认 10，含义是当系统有 10%的脏页面（待写页面）时，唤醒 pdflush。

在 pdflush 调优过程中需要考虑写入的平滑性，每次写入的数量不能大起大落。写入的时间性，每次写操作的间隔长短不一，同时又要考虑 Lazy synchronization，不能有一点就写一点，从而丧失了优化的环境。

8.5.3　磁盘读写方式

磁盘读写的方式有很多种，分类也很多，本节主要就搜索引擎行业常见的一些读写方式进行讨论。

（1）mmap

mmap 可以将文件的部分或者全部映射到内存中，其后对内存某个位置的读写会产生一次缺页，操作系统调页后，完成一次访问，在对内存的写入后，不会马上同步到磁盘上，如果需要立即同步，则必须调用一次 msync 函数，该函数给 flag 参数指派 MS_SYNC 或者 MS_ASYNC 成为同步或者异步的操作。最后通过 munmap 结束文件的映射。

mmap 是大文件并发访问的一种常见方式，mmap 的使用还有很多参数，在使用中需要仔细斟酌和实验，这里不一一赘述。mmap 的主要好处是减少了内核空间向用户空间复制的开销，用户程序和操作系统共享这块缓冲区，同时一个文件中自然会形成一些访问的热点，而这些热点所在的区域自然的都会被操作系统调页，以使得后继的访问可以在内存中进行，我们在对文件的长期读写好像在内存中一样。更有利的是在多进程情况下，这块内存可以被共享读写，非常有利于多核的环境。另外 mmap 可以自定义和磁盘同步的策略，有利于写平滑和批量写。

mmap 实现简单，应用于随机读，随机写；多进程，多线程的文件读写场合。

（2）Raw Device

裸设备（Raw Device）是不使用文件系统（ext2, reiserfs）而直接进行磁盘读写的一种方式。裸设备一般需要先将磁盘 umount，并进行必要的自定义的格式化方式，

接下来直接使用 open,read,write,lseek,close 等命令进行磁盘读写。在 Raw Device 上进行开发需要自行实现数据的格式化，缓存策略等以适应具体应用的需要，通常 Oracle 这种商业数据库一般均使用 Raw Device 的方式。

Raw Device 可以实现很灵活的 sector size，例如 512 字节，1k，2k…64k 等，在小数据存储时，使用小的 sector 可以避免空间浪费，而一般的文件系统，只有一种 logical block size，如果是 4k，即便一个 1k 的文件，也需要 4k 的空间。同时有助于文件的连续存放，避免碎片，例如一个 640k 的文件，在通常的文件系统需要 160 个 block，而磁盘连续分配和删除后，使得 160 个 block 在物理上可能并不连续，在读取这个文件的内部实际上是一种随机读的状态，在自定义块大小的情况下，可以将该文件存放在 10 个 64k 大小中，顺序读写程度大大加强。本质上说，这是文件系统的单一 block size 和绕过系统进入磁盘的多样 sector size 的差别，给性能改善提供了广阔的空间。

这里我们增加一个小常识：磁盘分为磁道（track），每条 track 上有若干 sector，每个 sector 为 512 字节。为什么是 512 字节呢？这是 1956 年由 industry trade organization, International Disk Drive Equipment 和 Materials Association 三家机构确定的行业标准。随着时代的发展，512 字节的 sector 明显太小了，由于每个 sector 还要存放很多其他信息，因此增大 sector size 可以降低 sector 的数量，从而提高实际存储量，同时降低了差错校验等很多 CPU 计算量。但遗憾的是由于这个标准太根深蒂固，很多代码开发都默认了 512 这个固定值，现在要想改势比登天。因此在设计裸盘的读写时必须保持可变的 sector size 是 512 字节的倍数，以符合工业标准。

有研究表明[oracle raw device]，oracle 数据库在使用裸设备后性能提升 50%。Raw device 的使用开发难度大，适用于大批量，复杂环境下的，需要自定义磁盘访问的场合。

（3）DIRECT_IO

在文件打开（Open）操作中指定文件的模式为 O_DIRECT，则文件的写入和读取直接在用户空间的内存和物理磁盘发生交互，不会走内核空间的数据交互，节约了用户空间内存到内核空间内存的复制的开销，并且读取和写入都是同步的，即 read 或者 write 函数返回后，文件已经保证被写入磁盘。在进行 DirectIO 时，要注意用户空间内存和文件偏移必须和文件系统的逻辑块大小对齐，Linux2.4 内核下一般在 4K 字节上对齐，在 Linux2.6 下，在 512 字节上对齐即可。

Direct IO，一般需要程序自行实现文件读写缓存，正如我们此前提到过，读通常具有同步性、随机性，而写通常具有异步性、批量性，就好像地铁站出站口的通道总是比进站口的通道多，出站一般是批量的，而入站则是随机的。因此读缓存（队列）可以设置得较少，而写缓存可以设置得较大，并自行做好排序和归并的工作，这样才能把减少内存复制获得的价值体现出来。Direct IO 一般用于大量顺序 IO 写入的场合。

（4）AIO

异步 IO（AIO）在大量并发 IO，且不要求同步的场合（数据无先后依赖关系）下使用，例如搜索引擎的爬虫下载网页的过程，异步 IO 可以大大提高磁盘读取得性能，但编程稍显复杂，需要在发起 IO 后，在回调函数中完成读取字节后的处理，或者写入后成功与否的处理。Linux 异步 IO 还包含 lio_listio 的功能，可以打包一组 IO 同时提交，进一步提高数据吞吐量，以下是可能用到的一些函数，如表 8-9 所示。

表 8-9　关于异步 IO 的一些函数

函　数	说　明
aio_read	请求异步读操作
aio_error	检查异步请求的状态
aio_return	获得完成的异步请求的返回状态
aio_write	请求异步写操作
aio_suspend	挂起调用进程，直到一个或多个异步请求已经完成（或失败）
aio_cancel	取消异步 I/O 请求
lio_listio	发起一系列 I/O 操作

proc 文件系统包含了两个虚拟文件，它们可以用来对异步 I/O 的性能进行优化：

/proZzzc/sys/fs/aio-nr 文件提供了系统范围异步 I/O 请求现在的数目。

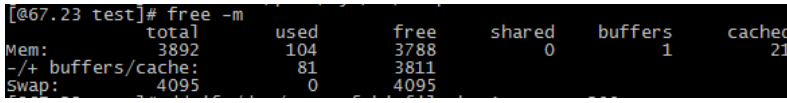
proc/sys/fs/aio-max-nr 文件是所允许的并发请求的最大个数。默认值 64k。

8.5.4　文件缓存问题

操作系统对文件读取时会进行缓存，缓存的对象是文件存放的 data block，在发生大文件更换时，通常使用 echo 3 > /proc/sys/vm/drop_caches 来回收文件缓存，我们通过一个有趣的实验来说明这个问题，每个执行命令前标有序号：

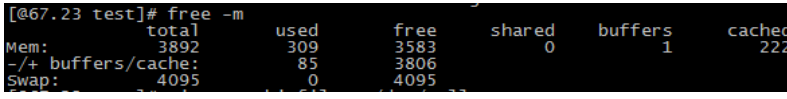
[1]echo 3 > /proc/sys/vm/drop_caches

[2]free –m

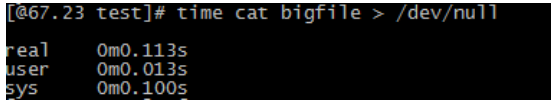


[3] dd if=/dev/zero of=bigfile bs=1M count=200　/*创建一个 200 兆的大文件*/

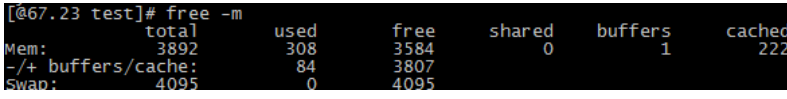
[4]free –m　/*因为 200 兆文件是一个写入的方式，因此文件内容均在缓存中。*/



[5] time cat bigfile > /dev/null　/*顺序读取一遍该文件，并计时，耗时仅 0.1 秒*/

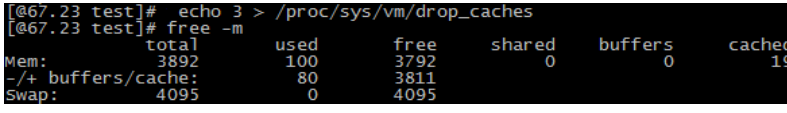


[6]free –m　/*考察清缓存前的内存状况*/

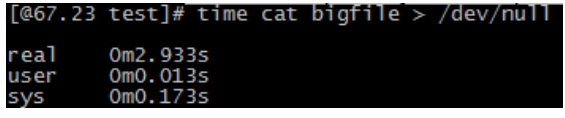


[7] echo 3 > /proc/sys/vm/drop_caches

[8] free –m　/*清除缓存后，使用内存从 308M 降低到 100M。*/



[9]　time cat bigfile > /dev/null　/*这时再读一遍 200M 文件耗时近 3 秒。*/



通过这个实验可以看出：首先，在文件写入并关闭后，文件缓存还是保持在系统中的，如果需要清除得手动执行。其次，清除缓存前后的文件顺序读取速度有接近 30 倍的差距，这一点在搜索引擎的很多地方都有应用，例如搜索引擎在索引更换时，需要执行一次 echo 3 > /proc/sys/vm/drop_caches 命令，将缓存在磁盘中的老索引内容全部清除。同时还需要做一些操作将部分新索引的内容换入内存中去。

8.5.5　5 分钟法则

1987 年，Jim Gray 和 Gianfranco Putzolu 推出了著名的 5 分钟法则[Gray 1987]，他们通过内存，硬盘的性能以及当时的成本，给出了这样的公式：BreakEvenIntervalinSeconds = (PagesPerMBofRAM /AccessesPerSecondPerDisk) ×（PricePerDiskDrive /PricePerMBofRAM）。并由该公式得到了 5 分钟左右的近似值，因此做出这样的判断，如果一个数据的访问周期在 5 分钟以内则存放在内存中，否则应该存放在硬盘中。

其中：

PagesPerMBofRAM：表示内存每兆字节的 Page 数，如果 page size = 4KB，则该值为 1MB/(4KB/page) = 256 page/MB。

AccessesPerSecondPerDisk：每块磁盘每秒支持的最大 IO 请求数，如表 8-10 所示为 250Page/(Second*Disk)。

（PagesPerMBofRAM /AccessesPerSecondPerDisk）：表示 1 兆的空间通过磁盘访问的方式所需要的秒数。256/200 = 1.25(Second*Disk)/MB，表示 1 兆字节需要 1.25 个盘秒来完成（类似工作任务按人月来做单位）。

PricePerDiskDrive：表示一块磁盘的成本，如表 8-10 所示为 48$/disk。

PricePerMBofRAM：表示每兆内存的代价，如表 8-10 所示为 0.024$/MB。

PricePerDiskDrive/ PricePerMBofRAM：表示用来买磁盘的钱可以买多少兆内存。48/0.024=2000MB /disk。

(PagesPerMBofRAM/AccessesPerSecondPerDisk)*(PricePerDiskDrive/PricePerMBofRAM)：表示用磁盘读取的耗费时间界限，2000MB *1.25=2500second，合 41min。如果一条数据的访问周期低于 41 分钟，则应该放在内存中，否则应该放在磁盘中。

对于这个公式我们可以这样理解。

1MB 永久存放在内存中的代价是 0.024$/MB，1MB 数据使用一次相当于花 0.024$换来的。

1MB 存放在硬盘中的代价是 1.25 (second*disk)/MB *48($/disk) / 1 个访问周期

=1.25 *48/2500= 0.024$，1MB 数据放在硬盘中，2500 秒访问一次，花 0.024$。

如果访问周期小于 2500 秒，放在硬盘中的代价大，周期大于 5100 秒，放在硬盘中就赚了。

不难计算，如果将固态磁盘看作内存，则可得到 48/(780/(32*1024))*(256/200)/60=43 分钟。如果将固态硬盘看作硬盘，则可得到 780/(50/(2*1024))*(256/35000)/60=4 分钟，这是对 5 分钟法则的很好的近似。另外修改每页的大小，也可以近似得到 5 分钟这个值，请读者自行计算。[Goetz Graefe 2008]对该问题在 2008 年做了回顾，得到的结论是：5 分钟法则在提出的 20 年后依然有效，但分化出两个 5 分钟法则，一个是把闪存看做硬盘，内存和闪存之间依然保持了 5 分钟法则，另外一个是在更大的 page size 情况下，闪存和硬盘依然保持了 5 分钟法则，并且对内存和磁盘的分界时间预测在未来的 20 年将会达到 5 小时左右。

如表 8-10 所示，数据来自 amazon.com, [Intel flash]，仅作举例之用。

表 8-10　内存，SSD 和硬盘的相关数据

项　　目	内存（现代某服务器内存）	SSD（Intel 某 SSD）	硬盘（希捷某服务器硬盘）
价格	50$	780$	48$
容量	2G	32G	500G
随机读性能	—	35000(IOPS)	200(IOPS)

笔者在工作中实际接触的服务器可以支持最大单核 16G 内存，64 个扩展槽，达到 1TB 的内存，但在举例过程中，仅考虑基本情况。有兴趣的读者可以根据工作环境的服务器情况做相应计算，看看得出的结果。

5 分钟法则有哪些具体的应用呢？搜索引擎的索引非常巨大，每次上线过程都要经过很多步骤，其中一条叫做 cache warming，即对那些常用的查询词，人工的对索引系统发起一次查询，使得在硬盘中的索引能够“上浮”到内存中，这样在下一次真实用户来查询时，无须磁盘读取，而直接从内存中读出，那么将多少常用词载入到内存中呢？什么样的查询频率的数据值得载入内存中呢？

事实上，计算机硬件发展到今天，CPU 是最快速的，内存的发展次之，最慢的是磁盘，因此体系架构也需要考虑硬件发展的现状，结合 5 分钟法则和运营成本的考虑，查询频率达到一定频繁程度的查询词的倒排表可以放到内存中，否则应该放在内存中，一般那些诸如电话号码、邮编、生僻姓名的查询词通常都在磁盘中，而常用的娱乐明星的姓名、影视歌曲等查询词的倒排表都被事先载入了内存。

除此之外，5 分钟法则还可以用来根据业务需要，指导硬件采购，有些数据库提供商往往能够通过一些科学方法来为客户的实际业务需要来配置最优的服务器。

8.6　涉及网络的优化方法

网络调优的范畴非常大，管理员通常只做内核参数的一般性调优，工程师通常只做简单的策略修改，重编译内核。但这并不是网络调优的全部，分布式环境，多机环境，具体业务之间的数据交互，非常复杂，知识点细碎。笔者有些经验，也很难做很好的归纳和总结，本节仅从搜索首页和结果页提速和 Web Server 架构两个点进行一些讨论。

8.6.1　搜索首页，结果页提速方法

抑制慢启动

慢启动算法在传输开始时和发生拥塞时启动。其主要目的是，优化网络带宽利用，在交互双方网络质量不明的情况下，阻止发送方发送大量的数据，如果不这样处理，会造成大量淹没中间网关从而引起数据丢失。

慢启动的初始拥塞窗口一般设置为 1 个 MSS(Maxium Segment Size)[cwnd=1]，这里解释一下 MSS 和 MTU 的关系，MTU 是链路层最大传输单位，以太网通常是 1500，MSS 通常是 1500-40=1460，其中 20 个字节为 TCP 包头，20 个字节为 IP 包头。MSS 大小一般很难修改，是在通信双方的 MSS 取最小值作为双方交互的 MSS 大小。在链路顺畅时，收端返回 ACK 小于重传时间的情况下，拥塞窗口呈现指数增长，数据交互逐渐变快，直到稳定。

通常来说慢启动对 short-lived connection 都是极为不利的，特别是搜索引擎首页和结果页的这种情况，根据[Jerry 2009]的统计数据，87%的搜索结果页 size 小于 10.5KB，大部分搜索结果页只有 7KB，如图 8-14 所示。

按照 MSS1460 计算，10.5KB 字节合 8 个报文，按照 cwnd= 3,6,12 的这个趋势，在无丢包的情况下，需要 2 个数据包发完，如果增加到 10，则 1 个包可以发完。

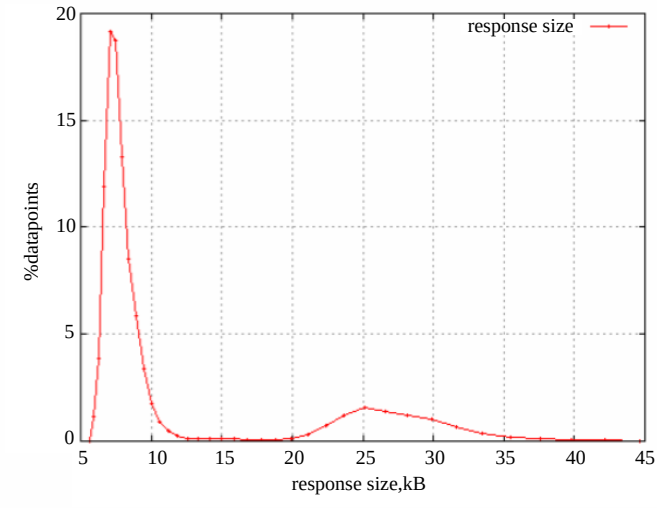


图 8-14　返回用户请求的服务端数据大小分布

参数的修改大多可以通过 sysctl 命令，但策略的修改一般需要修改内核，并重编译，优化时请注意拥有 root 权限，一般采用 patch 的方式编译内核，网上有很多教程，我们以修改默认窗口大小为例：在/net/ipv4/tcp_input.c 中，找到 tcp_init_cwnd 函数。

```
__u32 tcp_init_cwnd(struct tcp_sock *tp, struct dst_entry *dst)
{
      __u32 cwnd = (dst ? dst_metric(dst, RTAX_INITCWND) : 0);
     if (!cwnd) {
            if (tp->mss_cache > 1460)
                cwnd = 2;
            else
               cwnd = (tp->mss_cache > 1095) ? 3 : 4;
                                    //通常初始默认窗口大小为 3
     }
     return min_t(__u32, cwnd, tp->snd_cwnd_clamp);
}
```

做了修改，增加一个 sysctl_tcp_iw 方便用 sysctl 修改初始窗口参数，然后打一个 patch，如下：

```
@@ -747,7 +802,9 @@ __u32 tcp_init_cwnd(struct tcp_sock *tp,
 {
    __u32 cwnd = (dst ? dst_metric(dst, RTAX_INITCWND) : 0);
```

```
-    if (!cwnd) {                    //-表示要去掉的源代码
+    if (sysctl_tcp_iw) {            //+表示要增加的源代码
+              cwnd = sysctl_tcp_iw;
+    } else if (!cwnd) {
          if(tp->mss_cache > 1460) //没有+，-号表示和原来的代码一致，无变化
               cwnd = 2;
          else
@@ -757,15 +814,17 @@ __u32 tcp_init_cwnd(struct tcp_sock *tp,
 }
```

在命令行执行形如 patch –p0 < patch-2010-10-07-linux-tcp-patch 的命令。下面的操作可以参考内核编译的手册进行后续编译、配置，重启机器即可生效。

[Dukkipati 2010]给出了初始化窗口的实验，如图 8-15 所示，在窗口开到 16 时最佳，受到丢包率增大的影响，窗口开到 42 延迟反而增加，并推荐初始窗口定义为 10（约合 15K 字节）在高速网络的提升达到 10%，低速网络也略有改善，只付出了重传率增加 0.5%的代价。

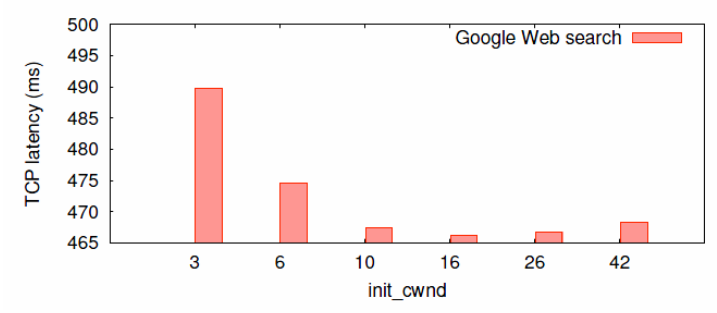


图 8-15　Google 搜索的初始窗口在不同情况下的延迟比较

读者可能会产生疑问，仅仅增加初始化发送窗口，如果接收窗口不配合，不采用发端的窗口建议，结果不是要打折扣吗？实验表明，大部分浏览器客户端都进行了很好的配合，除了 Linux 客户端，有兴趣的读者可以阅读[Dukkipati 2010]进一步了解。总之，在当今搜索引擎前端服务器，增加初始化拥塞窗口已经是非常普遍的调优手段。

快启动目前依然是研究热点，如何在降低数据交互的前期等待时间，同时更好地利用网络资源，降低拥塞，提出了很多想法。例如对每个交互的对象建立一个历史链路质量的记录，初始窗口根据以往经验来进行设置，采用 Packet Pacing 技术

[Vijay Sivaraman 2006]等。

强制快速重传

Linux 的超时重传默认是 3 秒，一旦发生丢包，会进行重传，同时超时重传扩大到 6 秒（3,6,9,12 这样的递增序列），该机制脱胎于上世纪的网络现状。目前随着技术改进，链路质量的提高，这一机制已不再适用，[Jerry 2009]指出大于 1 秒的 RTT 仅有大约 2.5%，而超时重传的概率仅有 0.8%~2.4%。

为了改善用户体验，可以采用强制快速重传技术来进行提速（Spurious SYN/SYN-ACK retransmissions），将超时重传时间强制设定为 1 秒。但这可能会导致 duplicate packets，可以通过修改 TS（tcp_sack 参数，链路质量良好的情况下一般设 0）或 DSACK（允许发多个 ACK，这里也用设 0 的方式）等方式来进行改进，如下：

```
sysctl -w net.ipv4.tcp_sack=0
sysctl -w net.ipv4.tcp_dsack=0
sysctl -p
```

还可以在/etc/sysctl.conf 中写入这些配置，然后重新启动网络守护程序/etc/rc.d/init.d/network restart，将这些配置生效。

强制重传的策略修改也需要修改内核，编译重启动才能生效。如果服务器还用于除 web 服务以外的其他功能，例如文件服务器、流媒体服务器等，这样的修改将是致命的。在进行这种修改时，需要特别小心，加强测试。

除了编译内核的方式，还有很多参数是可以很方便地调整，用 sysctl -a | grep net，查看，并存放于/proc/sys/net/ipv4/目录下。例如 TCP 的读写内存：net.ipv4.tcp_rmem, net.ipv4.tcp_wmem；keepactvie 时间：tcp_keepalive_time，拥塞窗口是否可调：net.ipv4.tcp_window_scaling。这些一般都需要调整的，默认参数大多不符合应用的实际需要。

8.6.2 Web Server 的架构选择

Web Server 的架构基本分为多线程和事件驱动两个基本形态，在实现上，将多线程和事件驱动相结合形成了流水线架构。在 Web Server 高并发、高吞吐率、低资源消耗和低延迟的要求下，Web Server 是事件驱动好，还是多线程好？论文[J. K. Ousterhout 1996]和[Rob 2003]各执一词，但就目前来看，兼而有之的流水线架构更为成熟。

在多线程的优点主要是容易理解，对资源的使用很容易饱和，缺点是上下文切换太多，假定线程代码段 1 份，10K 个线程，就是重复 10K 份，要想无上下文切换就需要 10K 个核。另一方面，资源占用过多，线程创建需要给线程栈分配内存，虽有办法降低这个消耗，但依然可观。最后，同步代码调试困难，正确性很难验证。

事件驱动的主要缺点是系统的资源（内存，CPU）不容易饱和使用，用户响应时间较长。优点是编程简单，可以支持很高的并发量，Linux2.6 内核开始支持的 epoll 就是事件驱动的典型实现，事件驱动可以支持很高的并发量。

在介绍架构之前我们先简单了解一下 C10K 问题。在硬件蓬勃发展的基础上，在时代发展的需要面前，越来越多的服务器面临了 C10K 问题[C10K]，即每台服务器应该能够同时满足 10K 个客户的并发访问需要。事实上，高并发在硬件上并不存在瓶颈，CPU 和内存均是可扩展的，出口带宽也绰绰有余，有研究表明即便并发在 10K 个用户，大部分情况下都不能把千兆带宽打满。因此良好的程序设计是关键。设计不够良好的程序，不具有可扩展性（scalability）。可扩展的含义是：一个能很好地在旧服务器处理 1000 个并发的吞吐量的程序，在 2 倍性能的新服务器上往往处理不了并发 2000 的吞吐量。

目前比较常见的架构分为 3 类：多线程架构和基于该架构的改进架构，事件驱动架构和基于该架构的改进架构，以及兼而有之的架构。这 3 类架构中我们选取 5 种进行逐个讨论：

- 多线程架构

简单的多线程架构

基于线程池的多线程架构

- 事件驱动架构

简单的事件驱动架构

- 兼而有之的架构

流水线的架构

多流水线的架构

在介绍架构之前，我们对架构好坏的评价做个分析，从两个层面来讨论：

用户角度：公平性，响应时间，吞吐率。其中响应时间有包括等待时间和处理时间。对人来说公平性是第一位的，如果一个架构是歧视性的，往往没有生命力，先进先出的策略是公认的公平的策略。每个用户都希望自己提交的任务可以尽快处理，但在耐受力上等待时间和处理时间有不同，如去银行办业务，等待 4 分钟办理业务 1 分钟的体验和等待 1 分钟办理业务 4 分钟的体验会不同。由于任务的处理时间和任务的复杂程度以及资源的竞争使用有关，因此架构主要考虑的是减少等待时间，减少资源的竞争。吞吐率是单位时间处理的请求数，这也是评价服务质量的重要指标。

系统角度：每指令需要耗费的指令周期 CPI，缓存命中率，资源的饱和使用程度，开发的难易程度，代码的可维护性等。在资源竞争程度高的架构上 CPI 会很高，在局部性不强的架构缓存命中率会很低，CPU 占用率低，磁盘 IO 低的架构是浪费的架构，开发难度大正确性很难保障的架构维护成本也会很大，因此架构越是自然，越是符合业务规律越是好架构，本节主要从这几个因素对架构做评价。

简单的多线程架构

如图 8-16 所示，是简单的多线程架构，螺旋线表示线程，合计 4 个线程。Accept Loop 承担的是 dispatch 任务的线程，其余线程处理用户请求。带箭头环线表示循环，带箭头曲线表示控制流。detached 表示线程执行完后即退出，不难看出，一个线程的从生到死只为一个 client 的 request 服务。

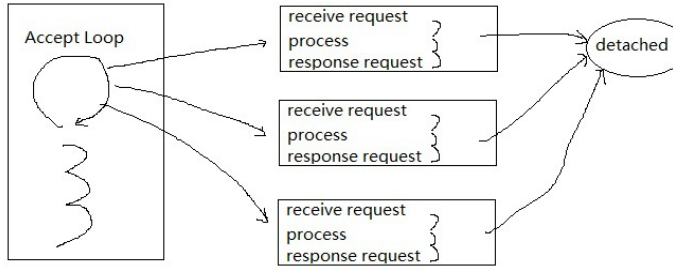


图 8-16　多线程架构示意图

简单的多线程架构的主要评价，如表 8-11 所示。

表 8-11　多线程架构评价

	公平性	响应时间	吞吐率	CPI	缓存命中率	开发难易程度	可维护性
简单多线程架构	差	短	低	高	低	易	高

对于该架构可做如下改进：

首先，是优化线程创建的开销。操作系统默认的进程初始栈空间，32 位操作系统为 1M，64 位操作系统为 2M（不同操作系统版本可能会有差异）。那么并发 10K 的线程可能需要 10G 内存，这是不可想象的，因此可以自行设定栈的大小和溢出区，代码如下：

```
size_t size = max(10*PAGE_SIZE，PTHREAD_STACK_MIN);
void* base = get_from_mmap(size);//从 mmap 分配的虚拟内存中取一块
ret = pthread_attr_setstack(&tattr, base,size);
```

假定我们的线程大部分情况下只需要 1 个 Page 的栈空间，我们用 mmap 的方式分配到的虚拟内存做自定义线程栈，合计 10 个 Page，另外 9 个 Page 看做是溢出区，如果线程的栈没有涨过 1 个 Page，那么这 9 个 Page 只是虚拟页，不会调实际物理内存页，因此可以看做是无开销，万一溢出了，只是多一个调页过程。

其次，读者很容易感觉到，一个 client 通常是短连接的，即便是 keep-alive 的形式，每个用户创建一个线程的代价还是太高，更严重的是，线程数量的大量增加，临界资源同步的成本大大增加，即锁的情况严重，是否可以让线程的创建保持在一个常量，控制争夺临界资源的线程数量呢？这就是线程池的思想，下面我们来看基于线程池的多线程架构。

基于线程池的多线程架构

如图 8-17 所示是线程池的架构。

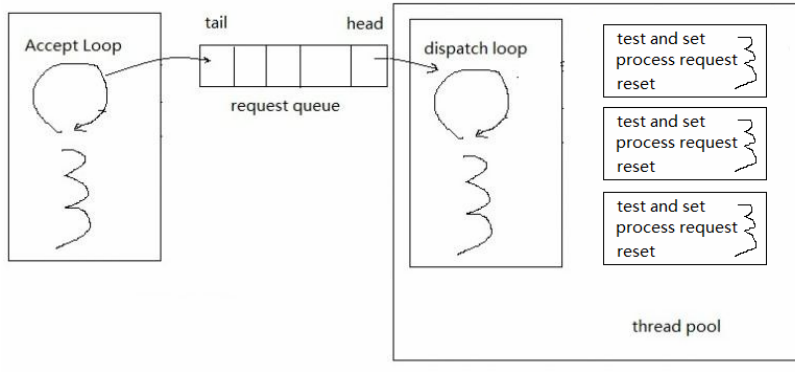


图 8-17　线程池的架构

系统在刚创建时，生成有限个线程，这些线程的生死都是随着系统的创建和退出的，和用户访问无关。在获得链接请求后，将用户的请求看做是一个消息，将其

插入到请求队列中，线程池的 dispatch loop 不断去将这个消息派发给一个闲置的线程进行处理，线程在处理完后等待 dispatch loop 派发新任务。线程池的实现方式有很多种，例如通过 test_and_set 和 reset 原子操作来进行等待和激活的操作。下面是部分实现代码。

```
int test_and_set(volatile int *s)     //当 s 非 1 时，返回非 1，否则返回 1，
                                      s 任何情况下均置 1
{      int r;
       __asm__ __volatile__(
                          "xchgl %0, %1 \n\t"  //xchgl 为一个原子操作
                           : "=r"(r), "=m"(*s)
                           : "0"(1), "m"(*s)
                           : "memory");
       return r;
}
线程一端执行如下伪码：
while (test_and_set(&S[thread_id].start) == 1){
                           //while 循环等待 dispatch loop 对 start 置 0
       block thread a while;
}
process(S[thread_id].request);
reset(S[thread_id].quit);             //通知 dispatch loop 处理结束。

dispatch loop 一端执行如下伪码：
for(thread_id = 1 to max_thread)
{
        if(test_and_set(&S[thread_id].quit) == 1)
                //如果 threadid 正忙，这里 tas 指令使用了 if 语句而不是 while
                continue;             //尝试下一个 threadid

        if(request_queue.not_empty()){
             S[thread_id].request = request_queue.deque();
            reset(&S[thread_id].start);//对 start 置零，解除线程等待条件
        }
}
```

基于线程池的多线程架构在并发线程的数量上大大减少，上下文切换相应减少；线程创建的数量与客户请求无关，创建成本减少；同时线程池的方式保证了按先进先出的次序进行响应，这一点很关键。一方面避免了饥饿，每个请求总能在有限的时间内完成（排在前面的请求处理完），一方面避免了资源争用，资源饱和使用的程度大大加强，使系统的吞吐率（throughput，单位时间交互的数据量)大大提高。但

付出了排队的成本，使请求的响应时间大大提高，这是线程池架构主要的问题。

我们日常生活中在银行办理业务，就可以看做是线程池模型的例子，排队机就是 request queue，叫号系统就是 dispatch loop，自动将到号的用户分配给一个空闲的客服，客服就是每个处理线程，客服在处理完业务后，按键进入闲状态，等待叫号系统安排。银行的这种设置显然是基于吞吐率等资源优化的角度考虑的，和简单多线程方式那种资源抢占式的无序相比，基于线程池的排队处理看上去更加优化。其评价如表 8-12 所示。

表 8-12　基于线程池的多线程架构评价

	公平性	响应时间	吞吐率	CPI	缓存命中率	开发难易程度	可维护性
基于线程池的多线程架构	好	短	中	高	低	中	高

简单的事件驱动架构

说事件驱动就不能不说 epoll——时下最流行的事件驱动最佳方案。在介绍 epoll 之前，先区分两个概念：边缘触发通知（edge-triggered readiness notification）和条件触发通知（level-triggered readiness notification）。

边缘触发：在应用程序传给内核一个文件描述符（FD）后（一个 SOCKET 链接可以看做是一个 FD），只有当该 FD 从 not ready 切换到 ready 状态（有字节可读）时，内核会通知这个状态变化，而不会通知是否已经读完，也就是后面需要由应用程序持续进行读取，直到读到 EWOULDBLOCK 为止，否则这个 SOCKET 的状态就一直保持在 ready 上，应用程序没有持续读到 EWOULDBLOCK，出现了交互的僵死，双方都不明对方的状态。边缘触发读通知，也有称作为准备状态改变通知（readiness change notification)。

条件触发：[J Lemon 2001]首次提出该术语，和边缘触发不同，条件触发的含义是“there is unread data in the buffer”，只要在 SOCKET 上还有未读完的数据，就会给出条件触发通知，通知应用程序有数据可读。

epoll 方案在业界被普遍采用之前，最早是 select 方法，但由于可扩展性等问题，又推出了 poll 的方法。然而无论是 Select 还是 Poll 在连接数增加时，性能急剧下降。这有两方面的原因。首先操作系统面对每次的 select/poll 操作，都需要重新建立一个当前线程的关心事件列表，并把线程挂在这个复杂的等待队列上，这是相当耗时的。其次，应用软件在 select/poll 返回后也需要对传入的句柄列表做一次扫描

来 dispatch，这也是很耗时的。这两件事都是和并发数相关，而 I/O 事件的密度也和并发数相关，导致 CPU 占用率和并发数近似成 O(n2)的关系[C10K 问题]。

我们先从一个图来看什么是事件驱动的架构，该架构也称作 Single-Process Event Driven 的架构，如图 8-18 所示。

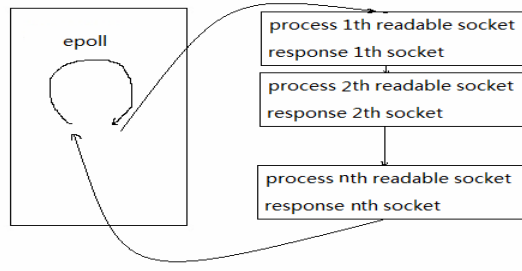


图 8-18　单进程的事件驱动架构

图 8-18 是一个最简单的单进程事件驱动模型，在进程中，epoll 方法轮询获取可读的 socket 列表，每个 socket 上的数据都是用户的 request。在获取了这些 socket 后，依次进行处理，并返回给用户端响应的数据。

在用法上 epoll 和 select,poll 类似，[C10K 问题]给出了一个简单的例子：

```
struct epoll_event ev, *events;
int kdpfd = epoll_create(100);
ev.events = EPOLLIN | EPOLLET;       // EPOLLET，指定了边缘触发
ev.data.fd =listener;
epoll_ctl(kdpfd, EPOLL_CTL_ADD, listener, &ev);
for(;;) {
   nfds = epoll_wait(kdpfd,events,maxevents,-1);//和 selec，poll 类似
   for(n = 0; n < nfds; ++n) {
     if(events[n].data.fd == listener) {
       client = accept(listener, (struct sockaddr*) &local,&addrlen);
          if(client < 0){
              perror("accept");
              continue;
          }
          else{
             setnonblocking(client);
             ev.events = EPOLLIN | EPOLLET;
             ev.data.fd = client;
             if(epoll_ctl(kdpfd,EPOLL_CTL_ADD,client,&ev)<0){
```

```
                fprintf(stderr, "epoll error: fd=%d”,client);
                return -1;
            }
        }
    else{
          do_use_fd(events[n].data.fd);//读或者写操作
    }
  }
}
```

epoll 的操作十分简单，一共就 4 个 API：epoll_create, epoll_ctl, epoll_wait 和 close，使用方便，本文不再赘述。简单事件驱动架构的基本评价，如表 8-13 所示。

表 8-13　简单事件驱动架构评价

	公平性	响应时间	吞吐率	CPI	缓存命中率	开发难易程度	可维护性
简单事件驱动架构	好	长	低	高	低	易	高

单进程的事件驱动架构因为是单进程，所以是一个无锁的环境，计算和存储资源不存在争用，当计算和存储资源性能提升后整个架构具有很好的扩展性（scability），与此同时该架构还具有严格有序的性质，即保证先来的请求先处理完毕，无论从响应还是从处理都是严格有序的，例如去医院挂号看病如采用该架构，它保证不会出现后挂号的比先挂号的先看完病的情况，显然如果有医院这么操作必然破产。该架构在资源利用率，缓存命中率，响应时间等方面均不理想，主要有几个原因：（1）处理一个请求可能有多个上下文，因此依次响应用户请求就会在这些上下文上反复切换，程序的局部性不强。（2）资源的有序使用导致了闲置，例如在等待读盘的时候，CPU 闲置。针对这两个问题，结合多线程的特点，产生了分阶段的事件驱动的架构（Staged Event-Driven Architecture）和多流水线的分阶段事件驱动架构。

分阶段的事件驱动架构

在处理一个复杂任务时，往往需要很多工序，以某工厂流水线的工序为例，第一步，把螺丝钉部分锤入模板；第二步，把螺丝钉继续旋进木板；第三步，把木板的四角锯掉。如果 1 个工人来处理这 3 步的话，则需要在锤子、螺丝起和锯子 3 种工具上进行切换，并且要掌握 3 种技术，这显然是非常吃力的。如果有 3 个人，每个人都具有这种能力，流水线的流动速度也会有限。但假如这 3 个人分别拿着锤子，螺丝起和锯子，分别做 3 个工序，速度可以大大加快，并且由于这 3 个人术业专精可以分别优化。

分阶段的事件驱动架构就是在利用事件驱动架构的基础上，将多线程的每个线程包办全部工作，拆分成线程只处理一小块任务，线程的计算和空间局部性就会大大提高。分阶段的事件驱动架构如图 8-19 所示：

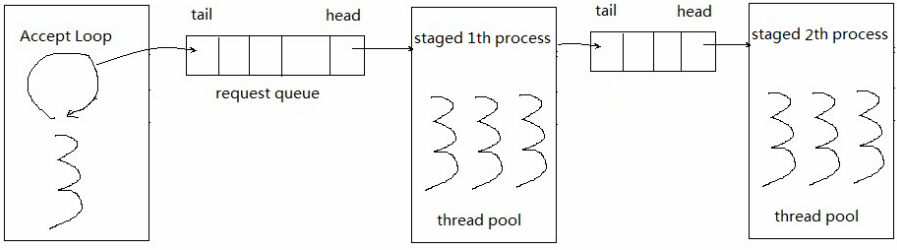


图 8-19　分阶段的事件驱动架构

上图是一个分两段的事件驱动架构，即单条流水线架构。一个复杂计算被分解成两个局部性强的计算阶段，在阶段内采用线程池充分发挥对资源的使用。跨阶段采用队列进行任务传递，由最后一个任务对用户请求做最后回复。这种拆分成流水线的架构的扩展性很强，可以将不同的阶段处理放在不同的服务器上。

流水线的设计需要注意这样几点：

（1）阶段的复杂性应大致一致。

从流水线理论上讲，每个阶段的处理时间相同是最佳的，如果某个阶段处理时间太长，就会出现后面的阶段饥饿，前面的阶段堵塞。短板流水线可以通过追加计算和内存资源，追加机器等方式来解决。

（2）阶段的划分尽可能从资源的角度出发。

例如第一阶段分词，只使用分词词典，第二阶段分类，只使用分类树，第三阶段决策，只使用决策树。阶段间不发生资源争用，资源只在阶段内争用。

（3）阶段的划分需要考虑剪枝。

传统的工艺流水线，通常会把破坏性强的工序提前，因为一旦出现次品就直接丢弃，后面的工作就省了，例如一次搜索结果的处理有这样 3 个工序，垃圾网页去除，死链去除，重复网页去除。假定这 3 个工序的复杂性依次提高，那么显然应该先过滤垃圾网页，将搜索结果数大大减少，丢掉的网页不再进行下面的计算。

（4）阶段的优化需要同步。

往往一个阶段过渡优化，在整条流失线上很难看出效果，需要对短板进行优化，并避免过渡优化。

多流水线的架构在此基础上进一步改进，相当于多个运行的实例，以求得对资源的最大利用。进一步提高对资源的饱和使用状况。表 8-14 所示为对分阶段事件驱动做一个基本评价。

表 8-14　分阶段事件驱动架构示意图

	公平性	响应时间	吞吐率	CPI	缓存命中率	开发难易程度	可维护性
分阶段事件驱动的架构	好	短	高	少	高	中	高

在 4 种基本架构介绍完毕后，从有序、锁和阻塞 3 个角度来看待他们的区别，如表 8-15 所示。

表 8-15　四种基本架构的区别

基本架构类型	从有序的角度看	从锁的角度看	从阻塞的角度看
多线程架构	无序	严重的锁（heavy lock）	无阻塞
线程池架构	响应有序	可控的锁（controlled lock）	无阻塞
简单事件驱动	严格有序	无锁（no lock）	有阻塞
分阶段事件驱动	响应有序	分阶段的锁（staged lock）	无阻塞

最后，表 8-16 例举目前已有的一些架构供读者参考[Farag Azzedin 2009]，有兴趣的读者可以下载这些源代码进一步阅读。

表 8-16　目前已有的一些参考架构

年　代	名　称	主要特点
1999	AMPED	单进程事件驱动
2001	Cohort Scheduling	在该服务器中，线程执行顺序通过重排，将类似的计算连续执行，提高缓存命中率
2001	SEDA	分阶段的事件驱动的流水线架构，每个阶段由一个线程池进行处理
2001	Multi-Accept	每次 Accept 接收多个请求的设计
2003	Cappriccio	伯克利大学研发的多线程包，其中线程链栈设计使得单机可以开到 10K 个线程
2005	Hybrid	事件驱动和多线程将结合的架构
2007	SYMPED	多条流水线的架构
2008	MEANS	应用 micro-thread 的事件驱动的架构

参考文献

[Andrei 2007]　Andrei Alexandrescu . Lock-Free Data Structures,2007.

[Berger 2000]Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson,P. R. 2000. Hoard: a scalable memory allocator for multithreaded applications. ASPLOS 2000.

[Branch elimination]http://cellperformance.beyond3d.com/articles/2006/04/benefits -to-branch-elimination.html.

[C10K]http://www.kegel.com/c10k.html.

[C10K 问题]搜狗实验室技术交流文档 Vol.1:1http://www.sogou.com/labs/report/1 -1.pdf.

[Cache pollution] http://en.wikipedia.org/wiki/Cache_pollution.

[Code optimization]http://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Code_ optimization/Pipeline.

[Compiler optimization]http://en.wikipedia.org/wiki/Compiler_optimization.

[Dukkipati 2010]N Dukkipati et al An argument for increasing TCP's initial congestion window，ACM SIGCOMM 2010.

[False Sharing]http://www.codeproject.com/KB/threads/FalseSharing.aspx.

[Farag Azzedin 2009] FaragAzzedin，Khalid Al-Issa A self-adapting Web server architecture: Towards higher performance and better utilization，2009.

[Goetz Graefe 2008] The Five-Minute Rule 20 Years Later, Communications of the ACM Vol. 52 No. 7, Pages 48-59 ,http://cacm.acm.org/magazines/2009/7/32091-the-five-minute-rule-20-years-later/fulltext.

[Gray 1987]Gray, Jim; Putzolu, Gianfranco R. (1987), "The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time", Proceedings of the ACM SIGMOD Conference, pp. 395–398.

[Intel2008] http://software.intel.com/en-us/articles/improved-linux-smp-scaling-user -directed-processor-affinity/.

[Intel false sharing]http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/.

[Intel flash]http://www.intel.com/design/flash/nand/extreme/index.htm.

[Intel optimization level] http://www.slideshare.net/hutuworm/peeling-the-onion-for -ipdc-forum09-mix-ver1.

[Intel opcode]http://ref.x86asm.net/geek.html#x0F00.

[J.D.Valois 1995]J.D.Valois. Lock-Free Data Structures. Phd thesis Rensselaer Polytechnic Institute,Department of Computer Science,1995.

[Jerry 2009]H.K. Jerry Chu .Tuning TCP Parameters for the 21st Century July 27,2009.

[J Lemon 2001] Jonathan Lemon. Kqueue: A generic and scalable event notification facility. Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, p.141-153, June 25-30, 2001. 14.

[J. K. Ousterhout 1996] J. K. Ousterhout. Why threads are a bad idea (for most purposes), Jan 1996. Presentation given at the USENIX Annual Technical Conference.

[Linux Symposium 2005] Proceedings of the Linux Symposium Volume Two July 20nd–23th, 2005 Ottawa, Ontario Canada.

[L Mich 2003]L Mich, M Franch, L Gaio. Evaluating and Designing Web Site Quality - IEEE MULTIMEDIA, 2003 - computer.org.

[M. Michael 1996]M. Michael and M. Scott. Simple, fast, and practical nonblocking and blocking concurrent queue algorithms. In Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing, pages 267–276, 1996.

[Oracle raw device]http://www.remote-dba.net/t_oracle_9i_administration_26_disk_raw_devices.htm.

[Pipelining]http://www-cs-faculty.stanford.edu/~eroberts/courses/soco/projects/2000-01/risc/pipelining/index.html.

[PForDelta code] http://blog.csdn.net/pennyliang/archive/2010/11/30/6045965.aspx.

[Processor affinity]http://en.wikipedia.org/wiki/Processor_affinity.

[RE Bryant 2001] Randal E. Bryant David R. O’Hallaron Computer systems: A programmer's perspective.

[Rob 2003]R.V.Behren and J.Condit and F.Zhou and G.C.Necula and E.Brewer,”Capriccio:Scalable Threads for internet Services,”Proceedings of the 9th ACM symposium on Operating Systems Principles(SOSP 03),Oct,2003,pp.268-281.

[Rob 2003]Rob von Behren, Jeremy Condit and Eric Brewer .Why Events Are A Bad Idea(for high-concurrency servers).

[V Sivaraman 2006] V Sivaraman, H Elgindy, D Moreland, D Ostry Packet pacing in short buffer optical packet switched networks. Proc. IEEE INFOCOM, 2006.

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Broadview®
www.broadview.com.cn

《走进搜索引擎（第 2 版）》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

您可以任意选择以下四种方式之一与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1．在线提交

登录**www.broadview.com.cn/13104**，填写本书的读者调查表。

2．电子邮件

您可以发邮件至**jsj@phei.com.cn**或**editor@broadview.com.cn**。

3．读者电话

您可以直接拨打我们的读者服务电话：**010-88254369**。

4．信件

您可以写信至如下地址：北京万寿路**173**信箱博文视点，邮编：**100036**。

您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

（2）您了解新书信息的途径、影响您购买图书的因素；

（3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想停止接收后续资讯，只需编写邮件“退订+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn即可取消服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（**www.broadview.com.cn**）上，或推荐至**CSDN.NET**等专业网站上发表，被发表的书评的作者将获得价值**50**元的博文视点图书奖励。

更多信息，请关注博文视点官方微博：**http://t.sina.com.cn/broadviewbj**。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱　博文视点（100036）　　电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明